AN APPROACH TO SOFTWARE SYSTEM MODULARIZATION BASED ON DATA AND TYPE BINDINGS

By

Roger M. Ogando

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1991

ACKNOWLEDGMENTS

This work is dedicated to Ms. Olga E. Rivera and my parents and grandmothers. I would like to express my deepest appreciation to my chairman, Prof. Stephen Yau, former cochairman Prof. Sying-Syang Liu, current cochairman Prof. Stephen Thebaut, Prof. Randy Chow, Prof. Justin Graver, and Prof. Jack Elzinga for their guidance and invaluable insight throughout this research. I am particularly indebted to Prof. Liu, Prof. Norman Wilde of the University of West Florida, Prof. Yau, and Prof. Thebaut for their financial support, supportive counsel and for dedicating long hours to discussions and reviews. I am also grateful to many fellow graduate and undergraduate students whose contributions in research and development made this thesis possible; in particular, I am grateful to my colleague, Abu-Bakr M. Taha, who made valuable contributions to this research project. In addition, my thanks go to many roommates, classmates, and people from all over the world who made an initially strange land feels like home. Finally, I would like to thank the PRA/OAS Fellowship for their additional financial support which they provided during my study.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

LIST OF TABLES vi

LIST OF FIGURES vii

ABSTRACT ix

CHAPTERS

1 INTRODUCTION 1

2 BACKGROUND 6

2.1 Design Recovery 6
2.1.1 Clustering Approaches 7
2.1.2 Program Slicing Approach 11
2.1.3 Program Dependence Approach 12
2.1.4 Knowledge-based Approach 13
2.2 Complexity Metrics 13
2.2.1 Modularity Metrics 14
2.2.2 Zage's Design Metrics 19
2.2.3 Cyclomatic Complexity 21
2.2.4 Stability Metrics 22

3 THE PROPOSED APPROACH 24

3.1 The Proposed Approach 25
3.2 Applicability of the Algorithms 37
3.3 Conditions for Best Results 38

4 TIME AND SPACE COMPLEXITY ANALYSIS 39

4.1 Algorithm 1. Globals-based Object Finder Complexity 39
4.2 Algorithm 2. Types-based Object Finder Complexity 41

5 EVALUATION OF THE APPROACH 44

5.1 Goals of the Evaluation Studies 45

5.2 Methodology of the Evaluation Studies 47
5.3 Primitive Metrics of Complexity 48
5.3.1 Definitions 49
5.3.2 Inter-group Complexity Factors 57
5.3.3 Intra-group Complexity Factors 61
5.3.4 Example of the Factors 68
5.3.5 Validation of the Factors 69
5.4 The Test Cases: Identified Objects, Clusters and Groups 80
5.4.1 Test Case 1: Name Cross-reference Program 80
5.4.2 Test Case 2: Algebraic Expression Evaluation Program 83
5.4.3 Test Case 3: Symbol Table Management for `Acx` 88
5.5 Comparison of Complexity 94
5.5.1 Test Case 1: Name Cross-reference Program 99
5.5.2 Test Case 2: Algebraic Expression Evaluation Program 101
5.5.3 Test Case 3: Symbol Table Management Program for `Acx` . . . 103
5.5.4 Summary and Conclusions of the Comparison 106

6 APPLICATIONS: DESIGN RECOVERY BASED ON THE APPROACH 110

7 A PROTOTYPE FOR THE PROPOSED APPROACH 115

7.1 The Object Finder: A GNU Emacs-based Design Recovery Tool . . . 115
7.2 Design Goals of the Object Finder 117
7.3 Design of the Object Finder 119
7.4 `Xobject`: Graphical User Interface 122

8 EXPERIENCE WITH THE OBJECT FINDER 124

8.1 Example of the Top-down Analysis: Name Cross-reference Program . 124
8.2 Comparison with *C++* Classes: Algebraic Expression Evaluation Program 129
8.3 Example of the Bottom-up Analysis: Name Cross-reference Program . 136

9 CONCLUSIONS AND FURTHER STUDY 140

APPENDIX A OBJECT FINDER PROTOTYPE USER'S MANUAL . . 145

A.1 Introduction 145
A.2 Operation 145
A.2.1 Basic Setup Commands 146
A.2.2 Object Finder Analysis 146
A.2.3 Display Analysis Results and Identified Objects 147
A.3 Buffer Structure and Files 151
A.3.1 System Buffer Structure 151
A.3.2 System Files 153

REFERENCES 155

BIOGRAPHICAL SKETCH 159

LIST OF TABLES

2.1 Information flows relation generated for the example. 18

3.1 Complexity index function for types in the "C" programming language. 34

5.1 Type size associated with several types. 56

5.2 Type size associated with variables of different types in the original version of the recursive descent expression parser. 70

5.3 Type size associated with variables of different types in version 1 of the recursive descent expression parser. 72

5.4 Statistics of the test case programs. 81

8.1 Statistics of the name cross-reference program. 124

8.2 Statistics of the algebraic expression evaluation program. 129

8.3 Comparison of candidate objects and "C++" classes. 131

LIST OF FIGURES

2.1 An example of information flow 17

2.2 Possible skeleton code for the example 17

3.1 Tree representation of types for complexity index function 35

5.1 Schematic illustrations of access pairs and data bindings 55

5.2 Example of the primitive complexity metrics factors 68

5.3 Identified objects in original version of recursive descent expression parser 71

5.4 Identified objects in version 1 of recursive descent expression parser . 73

5.5 Validation of the primitive complexity metrics factors 75

5.6 Groups based on objects identified in name cross-reference program . 82

5.7 Clusters found in the name cross-reference program by `basili` 82

5.8 Groups based on clusters found in name cross-reference program . . . 83

5.9 Groups based on types-based objects identified in algebraic expression evaluation program 84

5.10 Groups based on globals-based objects identified in algebraic expression evaluation program 86

5.11 Clusters found in algebraic expression evaluation program 86

5.12 Groups based on clusters found in algebraic expression evaluation program 87

5.13 Types-based candidate objects identified in symbol table management program 89

5.14 Globals-based candidate objects identified in symbol table management program 91

5.15 Groups based on types-based objects identified in symbol table management program . . . 92

5.16 Groups based on globals-based objects identified in symbol table management program . . . 94

5.17 Clusters found in symbol table management program . . . 95

5.18 Groups based on clusters found in symbol table management program 96

6.1 The object finder system flow . . . 111

7.1 The object finder conceptual model . . . 116

7.2 The object finder implementation outline . . . 118

7.3 A scanner of tokens . . . 121

7.4 Process to control the ANSI "C" cross-reference tool `acx` . . . 121

7.5 `Xobject` commands . . . 122

8.1 Candidate objects identified in the name cross-reference program . . . 126

8.2 Candidate objects identified in the name cross-reference program displayed by `xobject` . . . 128

8.3 Modified candidate objects in the name cross-reference program displayed by `xobject` . . . 130

8.4 Types-based candidate objects identified in the algebraic expression evaluation program . . . 132

8.5 Globals-based candidate objects identified in the algebraic expression evaluation program . . . 133

8.6 Candidate objects identified in algebraic expression evaluation program displayed by `xobject` . . . 135

8.7 Initial candidate object defined by the user in the name cross reference program . . . 137

8.8 Extended candidate object resulting from the data–routine analysis . 137

8.9 Final candidate object . . . 138

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment
of the Requirements for the Degree of Doctor of Philosophy

AN APPROACH TO SOFTWARE SYSTEM MODULARIZATION
BASED ON
DATA AND TYPE BINDINGS

By

Roger M. Ogando

August, 1991

Chairman: Stephen S. Yau
Major Department: Computer and Information Sciences

The maintenance of software systems usually begins with considerable effort spent in understanding their system structures. A system modularization defines a structure of the system in terms of the grouping of routines into modules within the system. This dissertation presents an approach to obtain a modularization of a program based on the object-like features found in the program.

While object-oriented methodologies for software design and development have only been clearly enunciated in the last few years, many object-like features such as data grouping, abstract data types, and inheritance have been in use for some time. In this dissertation, methodologies which aid in the recovery of the object-like features of a program written in a non object oriented programming language are explained. Two complementary methods for "object" identification are proposed which focus on data bindings and on type bindings in a program. The proposed approach looks for clusters of data, structures, and routines, that are analogous to the objects and object classes of object-oriented programming. The object finder is an interactive tool that combines the two methods while using human input to guide the object identification process. The experience of using the object finder and two evaluation studies of the object identification methods are also presented.

CHAPTER 1
INTRODUCTION

The maintenance of software systems usually begins with considerable effort spent in understanding their system structures and data. A system modularization defines a structure of the system in terms of the grouping of routines into modules within the system. This dissertation presents an approach to obtain a modularization of a program based on the object-like features found in the program. In modifying existing software, professional maintainers are almost unanimous in identifying the *understanding of system data* as one of their greatest challenges. Successful maintenance requires precise knowledge of the data items in the system, the ways these items are created and modified, and their relationships. Changing a program without a clear vision of its implicit data model is a very risky undertaking.

There seems to be little work that has explicitly addressed the problem of data understanding during software maintenance. A number of methodologies attempt to aid human program understanding of program constructs by cross referencing, by capturing dependencies [9, 46], by program slicing [13, 43], or by the ripple effect analysis of changes [50]. Other tools that have been proposed so far, such as those described by Ambras and O'Day [1], Biggerstaff [4], Kaiser et al. [16], Rich and Waters [32], and Yau and Liu [49], use knowledge-based approaches to provide inference capabilities. As a result, a user can derive additional information that may not be explicit in the program code.

Technologists in software design have made great progress in abstracting the ways computer programs use data. Most notable, perhaps, has been the emergence of the concept of *object-oriented design and development.* Booch defines an object as

"an entity whose behavior is characterized by the actions it suffers and that it requires of other objects" [5]. In practice, most objects are collections of data, together with the methods needed to access and manipulate those data.

Although object-oriented programming constructs are not directly supported in conventional programming languages such as "C" and "Ada," several object-like features, such as groupings of related data, abstract data types, and inheritance, have been in use for some time and may occur in an existing program. If such software needs to be maintained, it would be highly advantageous to identify the object-like features in the system. Knowledge of such "objects" would be important to:

1. Understand the system's design more precisely.
2. Facilitate reuse of existing "methods" contained in the system.
3. Avoid degrading the existing design by introducing unnecessary references to data that should be private to a given class of "objects."
4. Reengineer the system from a conventional programming language (such as "C") into an object-oriented language (such as "C++") to facilitate future enhancements.

We identified two important factors necessary for characterizing objects: global or persistent data [18] and the types of formal parameters and return values. Each factor will in turn originate a new algorithm for object identification. This dissertation presents these two algorithms for identifying object-like features in existing source code. One focuses on the data bindings between program components and the other on type bindings between program components. The two algorithms, as well as their implementations, are collectively known as the object finder.

The Globals-based Object Finder algorithm uses the information provided by persistent data structures to identify the objects in a program. Data bindings between system components have been previously used as the basis for clustering. In hierarchical clustering [14], for example, the elements chosen for grouping are the ones with the smallest "dissimilarity," that is to say, with the highest number of data bindings between elements. Our approach for globals-based object identification provides similar capabilities by grouping those routines which access a common global variable into "highly connected" objects. This algorithm handles procedural programming languages, such as "Ada," "C," "COBOL," "Pascal," or "FORTRAN," that provide scoping mechanisms that allow the definition of global variables and side effects on those global variables.

The Types-based Object Finder algorithm uses type binding as the basis for grouping routines into objects. This algorithm groups the routines according to the types of data used for return value and formal parameters of routines. Types-based object identification considers the "semantic" information provided by the types for the clustering. This is different from other clustering techniques based on semantic information such as conceptual clustering [35] which considers "light semantic" profiles about a system (from detailed cross reference information) during clustering. In the latter, a concept tree represents the common features of the members of a group, such as the names that a software unit uses ("names-used") and the names of the places where it is used ("user-names"). This algorithm handles those procedural programming languages which provide explicit type construction mechanisms.

This dissertation also presents the experience of using the object finder algorithms and an evaluation of the object finder algorithms through careful examination

of the results. Two studies were performed to evaluate the object identification algorithms. In study I, the evaluation consisted of comparing the groups (identified objects) identified using the object finder with those groups (clusters) identified using hierarchical clustering [14]. Study II compared the identified objects found in a program with the object-oriented programming classes found in the object-oriented version of the program.

The comparison in study I was based on the complexity of the two partitionings resulting from each clustering technique. The measure of the complexity of the partitionings is similar to coupling and cohesion [6]. Thus, we defined a new set of complexity metrics factors called inter-group complexity, which measures the complexity of the interface of a group, and intra-group complexity, which measures the internal complexity of a group, to measure the complexity of a given group in a partitioning. For this evaluation, we instrumented the program with access pairs and data binding triples in order to measure those metrics. These newly defined complexity metrics factors were validated and the results are reported in this work. The comparison results are also reported in this dissertation.

In study II, a "C" program, translated from a "C++" program, is used to compare the objects identified in the "C" program with the classes found in the "C++" version of the program. This example also illustrates some of the design recovery capabilities of the object finder when abstracting the underlying structure of a system.

The remainder of the dissertation is organized as follows: Chapter 2 provides a brief overview of design recovery and complexity metrics. Chapter 3 introduces the new approach, by way of several examples, for modularizing an existing program by identifying the object-like features found in the program. Chapter 4 discusses the

time and space complexity of the new approach. Chapter 5 presents the evaluation results of the proposed approach, using complexity metrics, and discusses the new set of complexity metrics factors and their validation. Chapter 6 discusses an application of the proposed approach in software systems design recovery and introduces a new approach for design recovery either top-down or bottom-up. Chapter 7 outlines the implementation of a prototype designed to demonstrate the flexibility and portability of the design of the prototype. Chapter 8 discusses the experience of using the approach to modularize and recover the design information from several programs. The most important limitation of the proposed approach is that it is currently restricted to two criteria of object identification. Many other criteria are suggested for the future study including object-oriented principles, such as the classification of the objects in the application, organization of the objects, and identification of operations on the objects. More experimentation is also needed to further evaluate the object identification approach, and other metrics factors should be added to measure the "object-orientedness" of the recovered design. The conclusions and the future study are discussed in Chapter 9.

CHAPTER 2
BACKGROUND

This chapter summarizes some relevant background information on design recovery and complexity metrics.

2.1 Design Recovery

Design recovery is defined by Biggerstaff [4] as recreating design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domain. Design recovery must generate all of the information required for a person to understand the functionality of a program, how it accomplishes its responsibilities, and why it performs that functionality.

Design recovery is common and critical throughout the software life cycle. The developer of new software usually has to spend a great deal of time trying to understand the structure of similar systems and system components. The software maintainer spends most of his or her time studying a system's structure to understand the nature and effect of a requested change. Without fully understanding the similar systems, the developer may not harvest the reusable components and reusable designs. Without fully understanding the program to be maintained, the maintainer may conduct inefficient or incorrect program modification.

Without automated techniques, design recovery is difficult and time consuming because the source code does not usually contain much of the original design information, changes are not adequately documented, there is no change stability, and there are ripple effects when making changes. Furthermore, large scale software worsen

these difficulties. Thus, automated supports for the understanding process are very desirable.

In the following sections we survey the current approaches to design recovery.

2.1.1 Clustering Approaches

The main characteristics of the clustering approach are that it shows the structural analysis of large software systems via adequate clustering techniques and the retrieval of high level structural information from the existing code [23]. The goal is to group routines into modules within the systems to reflect the modules defined by the developer.

Clustering is the analysis of the interface between the components of a system. It helps to determine the modularization that those interfaces define. The modules defined by this analysis are called *clusters*.

Clustering techniques with data bindings have been well studied, but the technique which involves the type bindings approach has only recently been published. Our object identification approach falls under the latter of these clustering techniques. In the following two sections, the data binding approach and the type binding approach are examined.

2.1.1.1 Data binding

In this section, we explain in detail the clustering techniques, based on data bindings, used in deriving a system's clusters. Later, we use these clusters to compare the grouping of the routines derived by the object finder, with those groups derived by these clustering techniques.

Data binding [2, 41, 14] reflects the interaction among the components of a system; it has been previously used for module interaction metrics [2]. In their work, Hutchens and Basili [14] use data bindings to measure the interface between

components of a system and to derive system clusters. For example, assume there are two procedures, p_1 and p_2, and a variable x in a program. When procedures p_1 and p_2 and the variable x are in the same static scope, whether the procedures access the variable or not, this is called a potential data binding, denoted as (p_1, x, p_2) because it reflects the possibility of data interaction between the two procedures. If both procedures access variable x, then there exists a used data binding. More work is required to calculate the used data binding than the potential data binding, but the former reflects a similarity between p_1 and p_2. If procedure p_1 assigns a value to variable x and procedure p_2 references x, this will reflect a flow of information from p_1 to p_2 via the variable x. This is called actual data binding and it is difficult to calculate since a distinction between reference and assignment must be maintained. Besides these bindings, control flow data binding is another kind of data binding, but it requires considerably more computation effort than actual data binding, because static data flow analysis must be performed.

After the data binding information is available, the system components can be grouped based on the strength of their relationships with each other [3]. This is derived by some specialized clustering techniques. The use of data bindings in clustering analysis provides meaningful pictures of high level system interactions and even system "fingerprints," which are descriptive analogies to galactic star systems. Next, we summarize Hutchens and Basili's hierarchical clustering modularization approach [14] and the current implementation [21] of their approach.

The nature of Hutchens and Basili's clustering algorithms is bottom-up or agglomerative since they iteratively create larger and larger groups, until the elements have coalesced into a single cluster. Thus, this is a module composition technique. The elements chosen for grouping are the ones with the smallest dissimilarity. There

are several methods for computing the dissimilarity [14]; the clustering algorithms used by Hutchens and Basili correspond to the "single-link" algorithm [14] which takes the smallest dissimilarity between the elements of each pair of newly formed clusters as the new coefficient between them.

The first step in hierarchical clustering is to abstract the data to obtain a binding matrix which represents the number of data bindings between any two components of the system. This matrix is a symmetric matrix. The current implementation considers all levels of data binding up to actual data bindings.

The second step is to obtain a *dissimilarity matrix* using one of two alternative methods:

1. *Recomputed Bindings*: based on the percentage of the bindings that connect to either of two components of the system. The dissimilarity matrix p is defined by $d(i,j) = p(i,j) = (sumi + sumj - 2b(i,j))/(sumi + sumj - b(i,j))$ where $sumi$ is the number of data bindings in which component i occurs and $sumj$ is the number of data bindings in which component j occurs. In this case, $p(i,j)$ is the probability that "a random data binding chosen from the union of all bindings associated with i or j is not in the intersection of all bindings associated with i or j" [14].

2. *Expected Bindings*: a weight is assigned to each binding level relative to the total number of elements under consideration in a given iteration. The dissimilarity matrix p is defined by $d(i,j) = (k/(n-1))/bind(i,j)$ where n is the number of elements under consideration and k is the number of bindings involving either element i or element j. Thus, one would expect $k/(n-1)$ of the bindings to be between i and j.

The current implementation of this clustering technique uses expected bindings to compute the dissimilarity matrix.

During the third step, the new clusters are formed by grouping together those elements whose dissimilarity is the smallest. Then, a new binding matrix is obtained from the clusters and the process starts again from this new binding matrix.

Hutchens and Basili summarize the problem of using data bindings only for modularization as that "whenever a module that defines an abstract data type and has no local data that is shared among the operations on the type, [the module] will not be located using this method" [14]. They explain that this is due to the fact that there is no direct data binding between the operations of the module and that all the interactions are indirect through the procedures that use the abstraction.

The algorithms for object identification in this dissertation present a solution to this problem that consists of using data types and establishing "relationships" between such types and the routines that use them for formal parameters or return values. Then, the abstract data type is revealed from the hiding imposed by the abstraction mechanisms.

2.1.1.2 Type binding

Type binding methodology [22] is the most recently published design recovery technique. It analyzes a conventional procedural programs in the context of an object-oriented paradigm.

Due to the gradual software paradigm progression from a purely procedural approach to an object-based approach and now to the object-oriented approach, object-like features, such as data grouping, abstract data type, and inheritance, already exist in conventional programming languages such as "C" and "Ada." It is very likely that the object-oriented concepts have been used in existing programs for

some time. Type binding methodology consists of identifying the objects in the conventional procedural programs in order to recover the underlying system structures [22]. The focus of this dissertation, the object finder, is a methodology that combines both the data binding approach and the type binding approach in modularizing a software system.

The object finder is somewhat analogous to other software clustering methods, but it is unique in searching for a particular kind of cluster which is similar to an abstract data type or object and cannot be clustered by a data binding methodology.

2.1.2 Program Slicing Approach

The concept of program slicing is originally discussed by Mark Weiser [43]. Weiser defines a slicing criterion as a pair (p, V), where p is a program point and V is a subset of the program's variables. In his work, a slice of program consists of all statements and predicates of the program that might affect the values of variables in V at point p. Weiser's work has been improved by Horwitz et al. [13], on the problem of interprocedural slicing — generating a slice of an entire program, where the slice crosses the boundaries of procedure calls.

The program slicing technique can help a programmer understand complicated source codes by isolating individual computation threads within a program. It can also aid in debugging and be used for automatic parallelization. Program slicing is also used for automatically integrating program variants; in this case, slices are used to compute a safe approximation to the change in behavior between a program P and a modified version of P and to help determine whether two different modifications to P interfere [13].

The main drawback of program slicing in recovering high level information is that this approach is oriented towards low level information abstraction. This

makes it difficult to understand high level system interaction and the overall system structure.

2.1.3 Program Dependence Approach

Program dependencies arise as the result of two separate effects. First, a dependence exists between two statements whenever a variable appearing in one of the statements may have an incorrect value if the statements were reversed. For example, given

```
A = B * C                   -- S1
D = A * E + 1               -- S2
```

S2 depends on S1, since executing S2 before S1 would result in S2 using an incorrect value for A. Dependencies of this type are called data dependencies. Second, a dependence exists between a statement and the predicate whose value immediately controls the execution of that statement. In the sequence

```
if(A) then                  -- S1
    B = C * D               -- S2
endif
```

S2 depends on predicate A since the value of A determines whether S2 is executed. Dependencies of this type are called control dependencies.

Program dependence analysis can be used in program slicing and in identifying reusable components for software maintenance [46]. Capturing the program dependencies can help program understanding to aid modification. On the other hand, this is limited to specific components of a program and it is does not facilitate high-level understanding of a program. However, one significant application of program dependence knowledge is in detection of parallelism and code optimization [9].

2.1.4 Knowledge-based Approach

Knowledge-based approaches [49, 1, 16, 32, 4] are very sophisticated and complicated general methodologies of design recovery. Their most distinguishing property as far as program understanding is concerned is that instead of just using the existing code, they use all the program information, including the source code, the documentation, the execution histories, program analysis results, etc.

The program information is expressed as patterns of program structures, problem domain structures, language structures, naming conventions, and so forth. It is stored in a central knowledge base which provide frameworks for the interpretation of the code.

A knowledge-based system is not a single tool, such as an editor or debugger. It is a collection of tools sharing a common knowledge base. It holds the promise of providing programmers a next-generation programming environment with the goal of dramatically improving the quality and productivity of software development [1]. However, knowledge-based approaches are still at the research stage. We consider that a more constrained tool, such as the object finder, provides immediate, more accurate knowledge about the high-level understanding of a program, directly from its source code.

2.2 Complexity Metrics

This section presents the complexity metrics considered for the evaluation of the object finder. We also explain the rationale for developing a new set of metrics to be used in this evaluation.

Software complexity is defined by Ramamoorthy [30] as "the degree of difficulty in analysis, testing, design, and implementation." However, our notion of software complexity is closer to the structure complexity metrics [15] which views the program

as a component of a larger system and focus on the interconnections of the system components.

The software metrics are classified as either *design* or *implementation* (code) metrics; design metrics measure the complexity of a design whereas implementation (code) metrics measure that of an implementation. The metrics chosen include design metrics such as the metrics of modularity [27] (cohesion and coupling) and implementation (code) metrics such as the cyclomatic complexity [25] of the program. Furthermore, we include a design-implementation metrics: the stability of programs [48].

In addition, we considered *code metrics*, which focus on the individual system components (procedures and modules) and require a detailed knowledge of their internal mechanisms, such as McCabe's cyclomatic complexity number [25]. We also consider, as indicated above, *structure metrics*, including Henry and Kafura's Information Flow [11] metrics and Yau and Collofello's Logical Stability [48] metric. Each of these metrics is measured by different characteristics of the program; that is to say, by evaluating the characteristics we may infer the degree of the metrics.

In the following sections we explain several of these metrics and the rationale for discarding each metric.

2.2.1 Modularity Metrics

Modularity metrics were initially defined by Myers [27]. They include two related aspects: *cohesion* (strength or binding) and the *coupling* of modules. Coupling is an indication of the level of interconnections between modules. A software component is said to exhibit a high degree of cohesion if the elements in that unit exhibit a high degree of "functional relatedness" [39]; that is to say, they exhibit functional unity [8]. The metrics of coupling among "modules" (either procedures/functions

in a non-object-oriented language and objects in the object finder) are measured by "structural fan-in/fan-out" degrees [6] and by "informational fan-in/fan-out" [11]. Since the data flow count includes procedure calls, informational fan-in/fan-out subsumes structural fan-in/fan-out [39].

Information flow [11] concepts are used to measure the module coupling between modules in a software system. These measures focus on the interface which connects system components. Myers [27] established six categories of coupling based on the data relationships among the modules. The information flow metrics can recognize two of these categories, including content coupling (refers to direct references between the modules) and common coupling (refers to the sharing of a global data structure). The information flow metrics is also used to measure the procedure and module complexity of a software system.

A measure of the "strength of the connections from module A to module B" [11] is:

$$(PEI(A) + PII(B)) * IP(A, B)$$

where $PEI(A)$ is the number of procedures exporting information from module A, $PII(B)$ is the number of procedures importing information into module B, and $IP(A, B)$ is the number of information paths between the procedures. Thus, the resulting metrics are a matrix of the coupling between any two modules in the software system.

The information needed to calculate this measure of coupling follows:

The information flow between modules depends on the information flow between procedures which are part of the module. A module is defined with respect to a data structure D in the program consisting of those procedures which either directly update D or directly retrieve information from D. Thus, examination of the global

flow in each module reveals the number of procedures in each module and all possible interconnections between the module procedures and the data structure.

There are several kinds of information flow between modules:

Definition 1 *There is a* global flow *of information from module A to module B through a global data structure D if A deposits (updates) information into D and B retrieves information from D.*

Definition 2 *There is a* local flow *of information from module A to module B if one or more of the following conditions hold:*

1. *if A calls B,*
2. *if B calls A and A returns a value to B, which B subsequently utilizes, or*
3. *if C calls both A and B passing an output value from A to B.*

Definition 3 *There is a* direct *local flow of information from module A to module B if condition (1) of Definition 2 holds for a local flow.*

Definition 4 *There is an* indirect *local flow of information from module A to module B if condition (2) or condition (3) of the Definition 2 holds for a local flow.*

Some examples of these information flows are illustrated in Figure 2.1. Figure 2.1 shows a simple system consisting of six modules (A,B,C,D,E,F), a data structure (D-S), and the connections among these components.

The possible skeleton code for this system is shown in Figure 2.2. As indicated in the skeleton code, module A retrieves information from DS and then calls B passing a parameter; module B then updates DS. C calls D, passing a parameter. Module D calls E with a parameter and E returns a value which D then uses and passes to F. The function of F is to update DS.

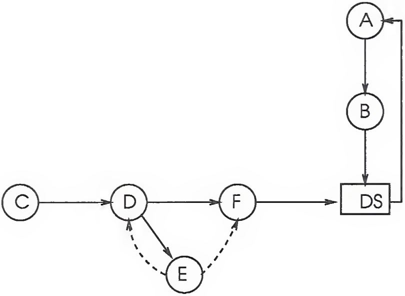

Figure 2.1. An example of information flow

```
type_ds DS;

void A()
{
  type_ds x;
  ...
  x = DS * 1;
  B(x)
  ...
}

void B(type_ds x)
{
  ...
  DS = x;
  ...
}

void C(type_ds p)
{
  ...
  D(p);
  ...
}
```

```
void D(type_ds p)
{
   type_ds q;
   ...
   q = E(p);
   F(p,q);
   ...
}

void E(type_ds p)
{
  ...
  return (p + 3);
  ...
}

void F(type_ds p, type_ds q)
{
  ...
  DS = p + q;
  ...
}
```

Figure 2.2. Possible skeleton code for the example

Table 2.1. Information flows relation generated for the example.

Direct local flows	$A \rightarrow B$
	$C \rightarrow D$
	$D \rightarrow E$
	$D \rightarrow F$
Indirect local flows	$E \rightarrow D$
	$E \rightarrow F$
Global flows	$B \rightarrow A$
	$F \rightarrow A$

The *information flow analysis* [11] process of a program consists of deriving the complete flow structure using a procedure-by-procedure analysis of the program.

The information flows for this example are summarized in Table 2.1. The direct local flows are simply due to the parameter passing. The indirect local flows are due to "side-effects" relationships between modules. The first indirect flow, from E to D, results when E returns a value (q) which D "uses" in its computation. The second indirect flow, from E to F, results when other information (q), that D receives from E, is passed unchanged to F. Finally, the global flow is due to information flow passing through the global data structure DS.

The next step is to compute the complexity of a module. The complexity of a module is the sum of the complexities of the procedures within the module.

The *complexity of a procedure* depends on two factors: the complexity of the procedure code and the complexity of the procedure's connections to its environment. A very simple length measure was used as an index of procedure code complexity [11]: the number of lines of text in the source code of the procedure (including embedded comments but not those preceding the procedure statement). The connections of a procedure to its environment are determined by the (informational) fan-in and fan-out of the procedure defined as follows:

Definition 5 *The* fan-in *of procedure A is the number of local flows into procedure A plus the number of data structures from which procedure A retrieves information.*

Definition 6 *The* fan-out *of procedure A is the number of local flows from procedure A plus the number of data structures which procedure A updates.*

However, in order to compute the complexity of the procedures which are included in a module, one should only consider local flows for the data structures associated with the module.

Finally, the complexity of a procedure is:

$$length * (fanin * fanout)^2$$

The term $(fan-in * fan-out)$ represents the total possible number of combinations of an input source to an output destination.

In conclusion, the complexity of a procedure contained in a specific module is computed using only the local flows for the data structure associated with that module. The complexity of a module is computed as the sum of the complexities of the procedures within the module. We use these complexity metrics concepts in defining a new set of complexity measure to evaluate the object finder.

2.2.2 Zage's Design Metrics

In the case of a structured design, Zage [51] developed a design quality metric $D(G)$ of the form

$$D(G) = k_1(D_e) + k_2(D_i)$$

In this equation, k_1 and k_2 are constants and D_e and D_i are, respectively, an external and an internal design quality component. D_e considers a module's *external* relationships to other modules in the software system, whereas D_i considers factors related to the *internal* structure.

The calculation of D_e and D_i is performed in two different stages of software design. D_e is based on information available during architectural design, whereas D_i is calculated after the detailed design is completed.

D_e is calculated for each module of a system and is comprised of two terms: one product related to the amount of data flowing through the module and another product giving the number of paths through the module.

$$D_e = (Weighted\ Inflows * Weighted\ Outflows) + (Fan\ In * Fan\ Out)$$

D_e appears to highlight stress points in architectural design. By redesigning these points, lower values of D_e were obtained [51] which, in effect, says that a reduction of D_e means a reduction on the coupling between modules.

The internal design metrics component D_i is calculated as follows:

$$D_i = w_1(CC) + w_2(DSM) + w_3(I/O)$$

where

CC	(Central Calls) are procedure or function invocations
DSM	(Data Structure Manipulations) are references to complex data types
I/O	(Input/Output) are external device accesses

and w_1, w_2, and w_3 are weighting factors.

The use of these three measures (CC, DSM, and I/O) is due to the desire to "choose a small set of measures which would be easy to collect and which would identify stress points to capture the essence of complexity" [51]. Their results indicate that stressing the data structure manipulation usage within modules gives excellent

results as a predictor of error-prone modules. The proposed values of the weighting factors are: $w_1 = 1$, $w_2 = 2.5$, and $w_3 = 1$. D_i was also better than cyclomatic complexity $v(G)$ and lines of code (LOC) as predictor of error-prone modules [51].

The design metrics were used as guidelines for the development of our new set of complexity metrics.

2.2.3 Cyclomatic Complexity

The computation of the metrics on cyclomatic complexity [25] of a program is based on the number of "connected" components, the number of edges, and the number of nodes in the program control graph. In practice, this metrics for structured programs consist of the number of predicates in a program plus one [25]. The conditionals are treated as contributing one for each predicate, thus we need to add two whenever there is a logical "and" and $N - 1$ whenever there is a case statement with N cases.

According to Shooman [37], McCabe has validated this metrics of cyclomatic complexity. McCabe concluded that, from a set of programs, those with a complex graph and a large $v(G)$ are often the trouble-prone programs.

The cyclomatic complexity of a program is derived by computing the number of conditions in each component (function). That is, McCabe's cyclomatic number for a collection of strongly connected graphs is determined from the union of those strongly connected graphs. The cyclomatic number of this union is computed to determine the cyclomatic number of the complete program.

The measure of complexity provided by these metrics focuses on individual system components. The complexity metrics which we must consider should focus on the structure of the system as well as on the high-level design information as criteria to determine the complexity of a modularization.

2.2.4 Stability Metrics

The metrics on program stability [48] are based on the stability of a module, defined as a measure of the resistance to the potential ripple effect from a modification of the module on other modules in the program. There are two sides to these metrics: the logical stability of a module, defined in terms of logical considerations, and the performance stability of a module, defined in terms of performance considerations. In our metrics study, we concentrate on the logical stability aspect of programs.

The logical stability of programs is defined in terms of a primitive subset of the maintenance activity, such as a change to a single variable definition in a module. The incremental approach to computing the logical stability metrics [48] of a program begins by computing the sets of interface variables which are affected by primitive modifications to variable definitions in a module by intramodule change propagation. In addition, for each interface variable, it is necessary to compute the set of modules which are involved in intermodule change propagation as a consequence of affecting the variable. Then, for each variable definition one must compute the set of modules which are involved in intermodule change propagation as a consequence of modifying the variable definition. Next, the individual complexity associated with each module is defined using McCabe's cyclomatic complexity. The potential ripple effect of a module is the probability that a variable definition will be chosen for modification times the complexity associated with the modules affected by such variable definition. Finally, the logical stability of a program is the inverse of this potential ripple effect of a primitive modification to a variable definition in a module.

We argue that our measure of complexity is based on similar grounds as the stability metrics. The stability metrics concentrate on the opposition to the propagation of the effect of primitive modifications by a software system. The propagation occurs

through the data transfer in the system. On the other hand, our metrics consider the relationship between components based on the links established by those data transfers.

CHAPTER 3
THE PROPOSED APPROACH

Our approach aids software maintenance by assisting the understanding process of the design of a software from its source code. The approach's output is a modularization of a program that consists of the collection of "objects" found in the program. This modularization of a system, that usually has no (or little) existing high level documentation, gives the maintainers an understanding of the structure of the system. Since the approach is based on the data and type of the system, it also assists the maintainers with understanding of the system data.

The proposed approach consists of a partial classification of the program elements (routines, types, and data items) that is meaningful in the context of the target program and its real world domain. The information required for this classification consists of the relationships between those program elements in term of data bindings and type bindings.

Two methods of object identification are used. The first is based on global and persistent data and establishes links to the routines that manipulate such data. The second method is based in the data types and establishes relationships between such types and the routines that used them for formal parameters and return values. Both methods result in sets of identified candidate objects.

The resulting candidate objects from both methods are not completely disjoint since they represent both object classes and instances of objects in the more classical sense. In addition, this allows the methods to capture the intentional "violations" made by the designer/implementor of the underlying design. The candidate objects

represent the structure of the program in terms of groups of routines implicit from the design and the relationships between those groups.

3.1 The Proposed Approach

The proposed approach consists of identifying the object-like features in conventional programming languages. Object-oriented constructs are not directly supported in conventional programming languages; however, several object-like features, such as groupings of related data and abstract data types, are found in those programming languages. The proposed approach identifies these groupings of data and abstract data types in terms of "objects." Most objects are collections of data, together with the methods needed to access and manipulate those data.

An "object," in a conventional programming language, can be identified as a collection of routines, types, and/or data items. The routines will implement the methods associated with the object, the types will structure the data they conceal or process, and the data items will represent or point to actual instances of the object class. Thus, we may characterize our candidate "objects" as tuples of three sets:

$$\text{Candidate Object} \;=\; (F, T, D)$$

where F is a set of routines, T is a set of types, and D is a set of data items. Any of these sets may be empty; ideally sets from distinct objects will not overlap so that a routine, type, or data item should not appear in more than one object.

A program contains routines, types, and data items. The proposed approach consists of a partial classification of these elements that is meaningful in the context of the target program and its real world domain. A large part of the information for this classification can come from analyzing the relationships between the components of the program, but human intervention or very carefully chosen heuristics will be needed to remove coincidental and meaningless relationships.

The identified candidate objects are not completely disjoint; there is some intentional fuzziness in the definition of candidate objects. It is often the case in a real program that the original implementor (or a later maintainer!) has violated the cleanness of the underlying design in a few instances, either from laziness or to gain efficiency. It would unnecessarily reduce the usefulness of object finding to reject out of hand any candidates that had small overlaps or violations of good information hiding practice.

Furthermore, the definition given does not distinguish clearly between the concept of an *object class* and the concept of an *object*. As a practical matter, in some cases it may be easier to first find the class and then its instances and in other cases to reverse this procedure. Thus, it is more convenient to treat the two together.

Two broad methods of object finding seem to be useful. The first is based on global and persistent (e.g. *static* in "C") data and establishes links to the routines that manipulate such data. The second methodology is based on data types and establishes relationships between such types and the routines that use them for formal parameters or return values. Without loss generality, we assume that all identifiers, such as routines, variables, and types, are distinguishable from their names.

The first method of object identification is given in Algorithm 1.

Algorithm 1 *Globals-based Object Finder*

Input : A program in a conventional programming language, such as Ada, COBOL, C, or FORTRAN, with scoping mechanisms.

Output : A collection of candidate objects.

Steps :

1. *For each global variable x (i.e., a variable shared by at least two routines), let $P(x)$ be the set of routines which directly uses x.*

2. *Considering each $P(x)$ as a node, construct a graph $G = (V, E)$ such that:*

$$\begin{aligned} V &= \{P(x) \mid x \textit{ is shared by at least two routines}\} \\ E &= \{\overline{P(x_1)P(x_2)} \mid P(x_1) \cap P(x_2) \neq \emptyset\}. \end{aligned}$$

3. *Construct a candidate object (F, T, D) from each strongly connected component (v, e) in G where*

$$\begin{aligned} F &= \cup_x\{P(x) \mid P(x) \in v\} \\ T &= \emptyset \\ D &= \cup_x\{x \mid P(x) \in v\} \end{aligned}$$

Example 1. Globals-based Object Finder—Single Stack/Queue:

To motivate this first method, take as an example a package in an Ada-like language for manipulating a single queue and a single stack of data with type `Element`. The package provides the interface routines so that the following three routines access a global data `STACK`:

```
procedure Push_S (X:Element);
     -- Push an element X to the Stack.
function Pop_S return Element;
     -- Pop the top element from the Stack.
function Is_Empty_S return Boolean;
     -- Return true if the Stack is empty.
```

and the following three routines access a global data `QUEUE`:

```
procedure Push_Q (X:Element);
     -- Push an element X to the Queue.
function Pop_Q return Element;
     -- Pop the front element from the Queue.
function Is_Empty_Q return Boolean;
     -- Return true if the Queue is empty.
```

If there is no other direct relation between these two groups, then clearly, `Push_S`, `Pop_S`, and `Is_Empty_S` belong to one candidate object and `Push_Q`, `Pop_Q`, and `Is_Empty_Q` belong to another. The Globals-based Object Finder given above would easily identify these two objects as the following two tuples:

$$(F_1, T_1, D_1) = (\{\texttt{Push\_S}, \texttt{Pop\_S}, \texttt{Is\_Empty\_S}\}, \emptyset, \{\texttt{Stack}\})$$

$$(F_2, T_2, D_2) = (\{\texttt{Push\_Q}, \texttt{Pop\_Q}, \texttt{Is\_Empty\_Q}\}, \emptyset, \{\texttt{Queue}\})$$

However, this method in many cases may produce objects which are "too big," since any routine that uses global data from two objects creates a link between them. Thus, a further stage of refinement will likely be necessary in which human intervention or heuristically guided search procedures improve the candidate objects by excluding offending routines or data items from the graph G.

The Globals-based Object Finder could utilize information other than the accesses to global variables by routines; in particular, it could also use the information about references and definitions of local variables and formal parameters. Then, a more detailed analysis of the internals of a routine will be required to obtain that information such as a semantic analysis to determine whether the references to local variables and formal parameters in a routine effectively access the same data item across invocations of the routine. Then, the routine could be made part of the group of routines that access that data.

The kind of analysis needed to obtain this knowledge will be developed in the future study of this dissertation. One suggestion is to use data flow analysis of the internals of routines to determine the "pattern" of accesses of local variables and formal parameters inside the routines. A pattern represents the uses and definitions of local variables and formal parameters in a routine similar to knowledge-based approaches in Section 2.1.4. These patterns could be used as a criteria to group

together those routines that exhibit similar patterns. For example, a pattern could be defined as the use of a local variable as an index in an array of elements. If this pattern is identified in a routine that has an array representation of an address table and in another routine with an array representation of a symbol table, we argue that both routines should be part of the same table indexing group.

The second method of object identification is given in Algorithm 2.

<u>*Algorithm 2*</u> *Types-based Object Finder*

Input : A program in a conventional programming language, such as Ada, COBOL, or C with data type abstraction mechanisms.

Output : A collection of candidate objects.

Steps :

1. *(Ordering) Define a topological order of all types in the program as follows:*
   (a) *If type x is used to define type y, then we say x is a "part of" y and y "contains" x, denoted by $x \ll y$.*
   (b) *$x \ll x$ is true.*
   (c) *If $x \ll y$ and $y \ll x$, then we say x is "equivalent" to y, denoted $x \equiv y$.*
   (d) *If $x \ll y$ and $y \ll z$, then $x \ll z$.*
2. *(Initial classification) Construct a relationship matrix $R(F,T)$ in which rows are routines and columns are types of formal parameters and return values. Initially, all entries of $R(F,T)$ are zeroes. An entry $R(f,t)$ is set to 1 if type t is a "part of" the type of a formal parameter or of a return value of routine f.*

3. *(Classification readjustment) For each row f of the matrix R, mark $R(f,t)$ as* 0 *if there exists any other type on the same row which "contains" type t and has been marked as* 1.

4. *(Grouping) Collect the routines into maximal groups based on sharing of types. Specifically, routines f_1 and f_2 are in the same group if there exists a type t such that $R(f_1,t) = R(f_2,t) = 1$.*

5. *Construct a candidate object (F,T,D) from each group where*

$$\begin{aligned} F &= \{f \mid \textit{the routine } f \textit{ is a member of the group}\} \\ T &= \{t \mid R(f,t) = 1 \textit{ for some } f \textit{ in } F\} \\ D &= \emptyset \end{aligned}$$

Again, in many cases the candidate objects created may be "too big." As can be seen in the following example, the culprit will often be a type that a human can easily identify as irrelevant to the objects being identified.

Example 2. Types-based Object Finder—Multiple Stacks/Queues:

In this example, there are four basic groups of routines. The first group manipulates complex numbers. The second group is related to multiple instances of stacks of complex numbers. The third group is related to multiple instances of queues of complex numbers. The fourth group involves routines that manipulate both stacks and queues. The four groups are specified in algebraic form as follows:

```
-- Group I:
   Construct (Real, Real) => Complex
      -- construct a complex from two reals
   "+" (Complex, Complex) => Complex -- plus
   "-" (Complex, Complex) => Complex -- minus
   "*" (Complex, Complex) => Complex -- multiplication
   "/" (Complex, Complex) => Complex -- division

-- Group II:
   Pop_S (Stack) => Stack x Complex
```

```
    --  remove the top element from a stack and return it
    Push_S (Stack, Complex) => Stack
       -- push a complex number onto a stack
    Is_Empty_S (Stack) => Boolean
       -- return true if Stack is empty

 -- Group III:
    Pop_Q (Queue) => Queue x Complex
       -- remove the head element from a queue and return it
    Push_Q (Queue, Complex) => Queue
       -- push a complex number onto a queue
    Is_Empty_Q (Queue) => Boolean
       -- return true if Queue is empty

 -- Group IV:
    Queue_to_Stack (Queue) => Stack -- convert a queue to a stack
    Stack_to_Queue (Stack) => Queue -- convert a stack to a queue
```

In this example, there are no global variables used. In applying Algorithm 2, we will develop the following matrix R:

Group ID	Routines	Complex	Real	Stack	Queue	Boolean
I	Construct	1	0			
	"+"	1				
	"-"	1				
	"*"	1				
	"/"	1				
II	Push_S	0		1		
	Pop_S	0		1		
	Is_Empty_S			1		1
III	Push_Q	0			1	
	Pop_Q	0			1	
	Is_Empty_Q				1	1
IV	Queue_to_Stack			1	1	
	Stack_to_Queue			1	1	

Blank entries (which will be marked as 0's according to Algorithm 2) indicate no direct relationship between the routine and the type. "0" means a "part of" relationship has been found by examining the internal data structures of the program.

In this example, at some point a complex will be found to contain real values and the stack and queue found to contain complex values.

The Types-based Object Finder will initially classify all the types and routines of groups II, III, and IV in a single large object because of the false links created by the `Boolean` type that link the stack and the queue. Some heuristics could be used to reduce such conflicts (e.g., eliminate primitive types, etc.), but it would also be a fairly easy task for a user to intervene at this point and identify `Complex`, `Stack`, and `Queue` as the objects of interest, provided that the data can be presented to him clearly.

There seems, however, to be no easy way to categorize group IV, which involves routines that operate on both `Queue` and `Stack` objects. A solution given by object-oriented design consists of a guideline which requires that a routine to be member of a class it must either access or modify data defined within the class [18]. Clearly, this indicates that a routine would be classified according to the type of its input formal parameters; then, routine `Queue_to_Stack` belongs to group II (the `Queue` candidate object) and routine `Stack_to_Queue` belongs to group III (the `Stack` candidate object).

Excluding group IV and type `Boolean`, the objects identified by Algorithm 2 would be listed as follows:

$$
\begin{aligned}
(F_1, T_1, D_1) &= (\{\texttt{Construct}, \texttt{"+"}, \texttt{"-"}, \texttt{"*"}, \texttt{"/"}\}, \{\texttt{Complex}\}, \emptyset) \\
(F_2, T_2, D_2) &= (\{\texttt{Push\_S}, \texttt{Pop\_S}, \texttt{Is\_Empty\_S}\}, \{\texttt{Stack}\}, \emptyset) \\
(F_3, T_3, D_3) &= (\{\texttt{Push\_Q}, \texttt{Pop\_Q}, \texttt{Is\_Empty\_Q}\}, \{\texttt{Queue}\}, \emptyset)
\end{aligned}
$$

The previously described topological ordering of types, in Step 1 of Algorithm 2, is appropriate whenever all types in a program are related in terms of the "part of" relationship defined above. This ordering represents the fact that a given type is

"part of" another type; thus, the latter type is "more important" than the former type when classifying routines into candidate objects. That is to say, a routine with formal parameters or return values with multiple types should be classified as part of the group of routines that manipulate data with the most important of those types, i.e., the type that was defined using the other types of the data manipulated by the routine.

The main problem with this type ordering scheme occurs when some types are not related by the "part of" relationship. In this case, the previous topological ordering of types does not completely characterize the relative importance of all the types; thus, the classification of routines using this type ordering scheme results in routines which may be classified under more than one type, such as the routines in group IV of Example 2. This problem can be handled by an alternative type ordering scheme based on the "complexity" of the data types in a system.

The *relative complexity type* ordering consists of ordering all the types in the program based on the "complexity" of the types. This complexity is expressed by a *complexity index function*, called CI, for a given type. Assume that the types in a system are represented using a tree that captures the "part of" relationship between types. Then, if type y is a "part of" type x, type x is an ancestor of type y and type y is a descendent of type x in the tree. An example of trees representing structure types is given in Figure 3.1 of Example 3. The complexity index of type t, denoted $CI(t, 0)$, is computed using the complexity index function in Table 3.1.

Given that the type of interest is the root of a tree representation of the type, the complexity index function recursively computes the complexity of the type as the sum of the path lengths (in number of arcs) from the root type to all its descendent types in the tree. In Table 3.1, the added complexity by primitive types is simply the length of the path, denoted d, from the root type to the primitive type. The complexity due to

Table 3.1. Complexity index function for types in the "C" programming language.

type t	$CI(t, d)$
primitive	d
pointer to primitive type y or user-defined type y	$d + (drf * CI(y, d+1))$
array of primitive type y or user-defined type y	$d + (dimension(t) * CI(y, d+1))$
`struct t { f1, f2, ..., fn}` and `f1`, `f2`, ..., `fn` are all base field types	$d + \sum_{j=1}^{n} CI(f_j, d+1)$
`struct t { f1, ..., fk, ..., fn}` and `f1`, ..., `fk` are base field types and `fk+1`, ..., `fn` are recursive field types	$d + S + f * S$ where $S = \sum_{j=1}^{k} CI(f_j, d+1)$ $f = (n-k)/n$

pointer types is modified by the *dereferencing factor*, called drf, which represents the pointer's complexity added to the complexity of the type; currently, the dereferencing factor may fluctuate between 1.5 and 2.0 depending on the system's use of pointer types. The complexity due to an array type, t, is modified by the number of elements in the array, i.e., its $dimension(t)$. The complexity due to a structure type consists of the sum of the complexity of its *base* fields types. The field types of a structure are of two categories: *recursive* field types are those which consists of either a pointer to the containing structure or essentially the same structure type as the containing structure (in "C", a `typedef` type); *base* field types are those which are not recursive types. In the presence of recursive field types, the complexity due to an structure type consists of the sum of the complexities of the base field types, denoted by S, plus a fraction of this complexity caused by the recursive field types.

Two types could be easily ordered by comparing their complexity indexes where a type A is more complex than type B iff $CI(B) < CI(A)$. Hence, type A is "more important" than type B in the topological ordering of types. A partial ordering of

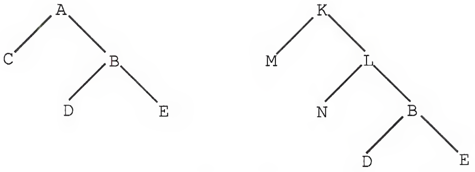

Figure 3.1. Tree representation of types for complexity index function

the types in a program is established by comparing the complexity indexes of all the types. Example 3 illustrates the computation of the complexity indexes.

Example 3. Type ordering based on the type complexity index function:

Consider two "C" data structures below where an identifier in capital letters denotes a type in the language and one in lowercase letters denotes a data structure field name. Assume that types `A`, `B`, `K`, and `L` are structure types and types `C`, `D`, `E`, `M`, and `N` are primitive types in the "C" programming language as follows:

```
struct A {
  C c;
  struct B {
    D d;
    E e;
  };
};
```

```
struct K {
  struct L {
    struct B {
      D d;
      E e;
    };
    N n;
  };
  M m;
};
```

The tree representations of those structures types, which captures the "part of" relationship among types, are shown in Figure 3.1. Then,

- The complexity indexes of types `A` and `B` in structure `A`, according to the complexity index function in Table 3.1, are:

$$CI(A,0) \;=\; 0 + CI(C,1) + CI(B,1)$$

$$
\begin{aligned}
&= 0+1+(1+CI(D,2)+CI(E,2)) \\
&= 0+1+(1+2+2)) \\
&= 6 \\
CI(B,0) &= 0+CI(D,1)+CI(E,1) \\
&= 0+1+1 \\
&= 2
\end{aligned}
$$

Type `A` is more complex than type `B`, thus type `A` is higher in the ordering of types than type `B`.

- The complexity index of type `K` is

$$
\begin{aligned}
CI(K,0) &= 0+CI(M,1)+CI(L,1) \\
&= 0+1+(1+CI(N,2)+CI(B,2)) \\
&= 0+1+(1+2+(2+CI(D,3)+CI(E,3))) \\
&= 0+1+(1+2+(2+3+3)) \\
&= 12
\end{aligned}
$$

Then, type `K` is more complex than type `A`. This type ordering scheme allows the comparison of two unrelated types to determine the most complex, and most important, of the types.

- The complexity indexes of type `B` by itself and within structures `A` and `L`, are equal since the complexity index depends on the structure that the type represents which is the same in all three cases.

An additional problem, which is not addressed by either of the two type ordering schemes, occurs when fields of an structure are conceptually "more important" than the structure, thus, the routines could be classified into objects according to the former. For example, consider another routine `Add_Top_to_Operand` in the multiple stacks/queues (Example 2) which implements the following functionality:

```
Complex Add_Top_to_Operand (Stack: st, Complex: op)
-- add the top element from a stack to another complex and return result
   Complex: result, top;

   top = Pop_S(st);
   result = "+"(top,op);
   return(result);
```

Using either of the two type ordering schemes, routine `Add_Top_to_Operand` would be classified under the `Stack` object. However, the functionality of the routine indicates that it should be classified under the `Complex` object since the routine implements operations on `Complex` entities of the program. The functionality of routines could be captured by a data flow analysis of the internals of a routine and by examination of the types of variables accessed inside the routines (including global variables, local variables, and formal parameters).

3.2 Applicability of the Algorithms

The applicability of these algorithms for object identification includes the kind of conventional, procedural programming language such as "Ada," "C," "COBOL," or "Pascal."

The Globals-based Object Finder handles those conventional programming languages as well as "FORTRAN." *Procedural programming languages* provide static and dynamic scoping mechanisms which allow the definition of global variables as well as local variables with respect to a particular scope level. The execution of the body of a function in the case of these programming languages results in *side effects* being observed on the values of the global variables [24].

The Types-based Object Finder handles procedural programming languages that provide a data abstraction mechanism which permits the construction of composite types using more primitive types. Clearly, typing mechanisms are also required for this algorithm. "FORTRAN" is one of these programming languages which does

not provide explicit type construction mechanisms, with the exception of arrays. The limitation of applicative languages, such as "LISP", is that they are not explicitly typed; in addition, their (abstract) data type construction mechanisms (e.g., list structure) are not currently handled by our analysis approach.

3.3 Conditions for Best Results

The proposed modularization approaches are particularly useful when the following conditions hold:

1. The program being maintained is written in a programming language that supports object-like features such as grouping of related data and abstract data types. Implicit abstract data types are identified by the approach of object identification. In the case of programming languages that explicitly support the syntactic specification of abstract data types, this modularization is used to define the relationships between the abstract data types defined in the system.

2. The Globals-based Object Finder requires that the program being maintained supports a static scoping mechanism. This allows the definition of global variables and the occurrence of side effects by the invocation of routines referencing the global variables.

3. The Types-based Object Finder requires that the program being maintained supports a data type abstraction mechanism that permits the construction of composite types using more primitive types.

CHAPTER 4
TIME AND SPACE COMPLEXITY ANALYSIS

In this chapter, the time and space complexities of the proposed approach are discussed. These complexities are independently analyzed for the two approaches of object identification.

4.1 Algorithm 1. Globals-based Object Finder Complexity

The time and space complexity of this algorithm follows:

Step 1. Build set $P(x)$ for each global variable x: Let N be the number of routines and g be the number of global variables in a system. Assume that the input to step 1 is a symbol table representation of a program. This symbol table consists of a sorted list of entries each of which contains an identifier's definition and references information. The time required to look up the references information about an identifier from the symbol table of a program is linearly proportional to the number of identifiers in the program; the time required is at most $O(g + N)$. Thus, for each global variable x, the time required to build the unordered set $P(x)$ of routines which directly access x is $O(g + N)$. For g global variables, the total time complexity of this step is $O(g(g + N))$. For real programs, N is usually larger than g, and the time complexity is $O(gN)$.

The space requirement for this step consists of the space to store the sets $P(x)$; the maximum size of a set $P(x)$ is $O(N)$. For g global variables, the space requirements is $O(gN)$.

Step 2. Construct graph G: Graph G construction consists of making each set $P(x)$ to be a node in the graph; the edges of the graph consists of the set intersection between two nodes (only its magnitude is important).

The data structure to store graph G consists of lists which represent the edges of the graph as follows: an edge $(g\_b_1\ g\_b_2\ P(g\_b_1) \cap P(g\_b_2))$ where g_b_1 and g_b_2 are global variables and $P(g\_b_1) \cap P(g\_b_2)$ is the set of common routines which is also represented as a list. The time required to construct one edge is the time required to obtain the intersection of unordered sets $P(g\_b_1)$ and $P(g\_b_2)$. The current implementation does not presort sets $P(x)$ before an intersection operation. Thus, the time is proportional to $|P(g\_b_1)||P(g\_b_2)|$ and the maximum time is $O(N^2)$. An improvement consists of presorting the sets $P(x)$. This improvement is possible since the kind of maintenance tasks (design recovery) which we consider do not involve changes to the connectivity of a program; then, the set $P(x)$, for global variable x, will remain unchanged after a modification. In this case, the time required to perform an intersection would be $O(N)$.

The total time required to construct all the edges is proportional to the number of global variables, g, in the program. The maximum number of edges is $O(g^2)$. Hence, the total time complexity is $O(g^2N^2)$.

Step 3. Construct candidate objects from strongly connected components: These candidate objects are obtained using a depth-first search algorithm [10, 42] for determining the connected components of the graph G; the complexity of such an algorithm is $O(M)$, where M is the number of edges in the graph which in turn is bound by the number of nodes in the graph as $O(g^2)$. This algorithm starts with some node of graph G. Then, we visit all the connected nodes in the order of a depth-first search, i.e., we walk, first, as far as possible into the graph without forming a cycle, and then

we return to the last bifurcation which had been ignored, and so on until we return to the node from which we started. We restart the procedure just described from some new node until all nodes have been visited.

Based on this analysis, the time complexity of the Globals-based Object Finder is $O(gN + g^2N^2 + g^2)$. The bounding time complexity is $O(g^2N^2)$.

The space complexity of this algorithm is also proportional to the space required to save graph G. The space required to save all the nodes in the graph is clearly proportional to the number of nodes (i.e., number of global variables) and the number of routines in the set $P(x)$ associated with a node. Since the maximum number of routines in a node is N, this space complexity is $O(gN)$. The space required to save all the edges in the graph is bound by the number of nodes and the intersection set corresponding to an edge, i.e., $O(g^2N^2)$. Then, the total space complexity is $O(gN + g^2N^2)$.

4.2 Algorithm 2. Types-based Object Finder Complexity

The time and space complexity of this algorithm only considers the type ordering scheme based on the "part of" relationship between types; it follows:

Step 1. Type ordering: The definition of a topological ordering is required for all the (abstract data) types used as types of formal parameters or return values of the routines in a program. The topological ordering of types defines an ordering of all the types in the program according to the "part of" relationship between any two types; i.e., if type t_1 is used to define type t_2, then we say that t_1 is a "part of" t_2. Assume the number of types used in a program, T, is proportional to the size of the program in lines of code L[1]. One algorithm for topological sort [38] has time complexity of $O(n^2)$ where n is the number of vertices in the graph. In our analysis, the number

[1] For real programs, T will be usually smaller than L.

of vertices is T. Then, the time complexity, using this topological sort, is bound by $O(T^2)$. Another algorithm for topological sort [17] has a total time complexity bounded by $(32m + 24n + 7b + 2c + 16a)$ where m is the number of input relations between types, n is the number of objects, a is the number of objects without no predecessors (primitive types), b is the number of tape records in input, and c is the number of tape records in output.

The topological ordering of types is stored as a tree which represents the "part of" relationship between types in a program. The "part of" relationship, between two types x and y, is stored as lists $(x \quad y)$ where type y is "part of" type x and type x "contains" type y. Given a list $(t_1\ t_2\ \ldots\ t_n)$, then type t_1 "contains" types t_2 through t_n. The maximum space required to save this tree is $O(T^2)$. This tree of types usually has a maximum of three or four levels in its branches.

Step 2. Initial classification: The time required to construct matrix R with N routines and T (abstract data) types is $O(TN)$. For real programs, T is usually smaller than N. Thus, the time required to construct matrix R is bound by $O(N)$.

Step 3. Classification readjustment: The time required for the classification readjustment is proportional to the number of routines, N, and the number of types, T, in matrix R. For a given routine in row r which has been marked with 1 in type t, the time required to determine whether there exists any other type s on the same row r which "contains" type t and has also been marked with 1, is proportional to the number of types T and to the time required to determine whether type s "contains" type t. The latter time is proportional to the number of types in the type ordering tree, and it is equivalent to the time to search for a path from type t to type s in the

subtree of the type ordering tree with root equal to s. The algorithm used to search for this path is a depth-first search [47] which has constant time complexity [38].

Hence, for a routine, the time required for the classification readjustment is $O(T)$. Given that we have N routines, the time required for the classification readjustment of matrix R is $O(NT)$.

Step 4. Grouping: The candidate objects are formed by collecting all routines, r, that share a type t; this is to say, for a given type in column t, a candidate object of routines, sharing type t, consists of all the routines, in row r, with (column) type t set to 1 after step 3. The time required to form these candidate objects is proportional to the number of (abstract data) types T and the number of routines N in the system. The time complexity of this step is $O(TN)$.

Based on this analysis, the time complexity of the Types-based Object Finder is $O(T^2 + N + NT + NT)$. As indicated above, for real programs T is usually small with respect to N, and the bounding time complexity is $O(max(T^2, NT))$.

The space complexity of this algorithm is clearly the space required to store matrix R plus the space required to store the tree representing the types ordering, i.e., $O(NT + T^2)$.

CHAPTER 5
EVALUATION OF THE APPROACH

This chapter presents some guidelines for an evaluation of the object identification methods. The goals of this evaluation are to compare the algorithms used for identifying objects with other existing modularization techniques in term of the complexity of the resulting modularization and to compare the resulting modularization in a conventional, procedural programming language with the "classes" found in an object-oriented version of the program.

The evaluation of the object finder algorithms consists of careful examination of the results of this approach. Two studies were used to evaluate the identified objects. In the first study, named study I, we compared the groups (based on the identified candidate objects) identified by the object finder with those groups (based on the clusters) identified with hierarchical clustering [14]. The comparison was based on the complexity of these two partitionings resulting from each analysis. The results of this study are presented in this chapter. In the second study, named study II, we compared the identified objects found in a program with the object-oriented programming classes found in the object-oriented version of the program. We explained the results of study II in Section 8.2.

A system's partitioning results from the system modularization, i.e., the grouping of routines into disjoint groups. The object finder and the hierarchical clustering technique [14] are different methods to obtain these partitionings.

Metrics of the complexity of software structures have been used as valuable management aids, important design tools, and as basic elements in research efforts to establish a quantitative foundation for comparing language constructs and design

methodologies [11]. In addition, module and interface metrics have been used in evaluating the modularization and the level of coupling of a system [14]. In study I, we continue to use metrics to evaluate the complexity of a system partitioning in terms of the complexity of the interfaces and the complexity of its components. These two views of complexity parallel coupling and cohesion. Conte et al. [7], measure coupling as the number of interconnections among modules and cohesion as the measure of the relationships of the elements within a module. A design with high degree of coupling and low module cohesion will contain more errors than a design with low module coupling and high module cohesion. Several primitive complexity metrics can be used to quantify the coupling and cohesion of a module. Then, several factors are used to quantify our complexity metrics.

5.1 Goals of the Evaluation Studies

The primary goal of the evaluation studies was to demonstrate that the algorithms used for identifying objects result in system partitionings less complex than other existing modularization methods. This chapter also includes the development of a new set of primitive factors that measured the complexity of a modularization of a system by characterizing the complexity of the corresponding partitioning.

The evaluation consisted of comparing the complexity of the candidate objects identified by the object finder and the complexity of the clusters defined using Basili's hierarchical clustering technique [14]. Another motivation for comparing the two approaches was to determine whether the object finder approach results effectively capture the structure of a software system. Since Basili's hierarchical clusters represent an experimentally validated initial approximation of the groups intended by the designer of the software system [14]; then, the results of the object finder should be less, or at least equally, complex than the hierarchical clustering approach results.

The chosen criteria to make the comparison was the complexity of each partitioning measured by complexity metrics similar to coupling and cohesion. According to Hutchens and Basili [14], the question of strength and coupling between elements of a given partitioning is largely still unresolved. Thus, a new set of primitive factors was developed to measure the complexity of the relationships between modules of a software system as well as the complexity of the relationships inside the modules. Furthermore, a measure of module strength in terms of the uniqueness of the types manipulated by the module was developed. That is to say, the new set of factors when applied to the partitioning of a system, measures the complexity of the interface between the system modules, the complexity inside the modules, and the strength of the components inside a module.

Other metrics [27, 25, 6, 11, 51, 48, 29] were considered for computing the complexity of the partitioning. They were not suitable for this purpose since the characteristics to be measured corresponds to the complexity of the interface (similar to coupling) and to the complexity of the relationships between elements inside a module (similar to cohesion). The object finder approach assumes that we identify candidate objects in programs which were originally designed without explicit object-oriented syntactic features even though we assumed that object-like features are present in the programs. Thus, metrics based on object-oriented syntactic features could not be used in the evaluation. Hence, we created a new set of primitive complexity metrics factors which measured three aspects of complexity: factors that measured the complexity of the interfaces between modules in a partitioning; factors that measured the complexity of the relationships between components inside modules, in terms of interactions between components in the modules; and primitive metrics factors that measured the strength of the relationships between components

in a module in terms of the similarity of data types manipulated by the components of the module.

5.2 Methodology of the Evaluation Studies

The methodology of the evaluation of the object finder included two studies: (1) study I consisted of comparing the groups identified by the object finder (candidate objects) with those groups identified by hierarchical clustering [14] (clusters), and (2) study II compared the identified candidate objects in a program with those classes found in the object-oriented version of the same program.

Several other evaluation studies are proposed for the future research including:

- Use of experts to compare the identified candidate objects in a system with the expert knowledge about the system.
- Use student academic projects as well as industrial-size programs to further evaluate the object finder algorithms.

The steps in study I were:

1. Identify example programs for the study. Three sample programs were identified for this study from the literature. Another industrial-size software system was considered for future evaluation studies.
2. Compute the identified candidate objects using the Top-down Analysis (Method 1 in Chapter 6) for a sample program. The result of this method is a partitioning of the system that consists of groups which corresponds to the identified candidate objects.
3. Compute the clusters in the sample program using Basili's hierarchical clustering technique [14]. The result of this technique is a partitioning of the system which normally corresponds to the top-level clusters of the system.

4. Compute the primitive complexity metrics factors for each partitioning above. In this step, we determined the primitive metrics factors for a partitioning of the system according to the definitions in Section 5.3.

5. Compare the complexity of the two partitionings using the results of the primitive complexity metrics factors for each partitioning.

The steps in study II are:

1. Identify example programs for the study. One program was chosen for this study from the literature.

2. Conversion of object-oriented code into non object-oriented code. For the comparison of identified objects and classes of an object-oriented version of a program, we require a program with two versions including an object-oriented version and an equivalent non object-oriented version. We choose an object-oriented program from the literature. Then, we derived a functionally equivalent version of this program using non object-oriented techniques.

3. Compare the identified objects with the classes.

The instrument and results of study II are reported in Section 8.2.

5.3 Primitive Metrics of Complexity

Assume that a system contains a large number of routines and data structures. Our complexity metrics factors measure the ability to partition [3] the system as to absorb as many relations between routines in a group as possible and thus leave few inter-group connections which results in less complex group interfaces. In addition, the complexity metrics measure the ability of reducing the number of relations between routines inside a group which results in less complex relations inside groups.

That is to say, we categorize what makes one partition "better" than another in spite of the fact that the connectivity of routines is always the same and only their group assignment changes from partitioning to partitioning. Beladi and Evangelisti define the degree of *connectivity* of a cluster as the number of "connections" between the elements of the cluster. Program routines and data structures are interconnected by routines invocations and references in software systems [3].

In the absence of a direct measure of the inter-group complexity and intra-group complexity which given a partitioning would compute these metrics, we developed the set of primitive complexity metrics factors which, we argue, measure inter-group and intra-group complexities, as well as intra-group strength, as demonstrated in Section 5.3.5. The union of all primitive metrics factors related to one kind of complexity (inter-group complexity, intra-group complexity, or intra-group strength) quantifies the corresponding complexity.

The following sections present important definitions and examples, the primitive factors of complexity, and a validation of these factors.

5.3.1 Definitions

Assume that all variables in a program have unique names. A use of a variable refers to either a definition (a value is assigned to a variable) or a reference (a variable's value is used) of the variable [12].

Definition 7 *A* global variable *is a variable directly used within at least two modules.*

Definition 8 *Let P be a module. There is a* module-global access pair (P, g) *if g is a global variable used within P.*

Synopsis: A module-global access pair (M, g) represents the fact that global variable g is used within module M.

<u>*Example 1*</u> *Module-global access pair* (M, g)

```
M()
{
 // g is used within M
}
```

<u>*Definition 9*</u> *Let* (Q, g) *be a module-global access pair. There is a* module-global indirect access pair (P, g) *if* $\exists$ *a module* Q *with a call to* P, $P(\ldots, g, \ldots)$ *such that the formal parameter of* P *corresponding to* g *is used within* P.

Synopsis: A module-global indirect access pair (M, g) represents the fact that global variable g is indirectly used by module M when there is a call to M with formal parameter g in module N which uses g.

<u>*Example 2*</u> *Module-global indirect access pair* (M, g)

```
N()                    M(a)
{                      {
 M(g)                   // a is used within M
}                      }
```

<u>*Definition 10*</u> *Let* P, Q *be two modules and* g *be a global variable in* P *and* Q. *There exists a* module-global access data binding *triple* (P, g, Q) *if* g *is defined within* P *and referenced within* Q.

Synopsis: A module-global access data binding triple (P, g, Q) represents the fact that module P defines the value of global variable g and module Q references the value of global variable g. It reflects the data relationship between modules P and Q, and the direction of the information flow (from P to Q).

<u>*Example 3*</u> *Module-global access data binding triple* (P, g, Q)

```
P()                    Q()
{                      {
 g<- // is defined       <-g // is referenced
}                      }
```

<u>*Definition 11*</u> *Let P, Q be two modules and x be a local variable in P. There is a* local-export data binding *triple (P, x, Q) if*

1. *x is defined within P before a call $Q(\ldots, x, \ldots)$ in P, and*

2. *the formal parameter of Q corresponding to x is referenced within Q.*

Synopsis: A local-export data binding triple (P, x, Q) represents the fact that module P defines the value of local variable x, and the corresponding formal parameter is referenced within module Q. It is based on the binding between the local variable and the corresponding formal parameter and the direction of the information flow (from P to Q).

<u>*Example 4*</u> *Local-export data binding triple (P, x, Q)*

```
P()                    Q(a:[by reference,by value-result, by value])
{ x:local              {
 x<- // is defined       <-a // is referenced
 Q(x)
}                      }
```

<u>*Definition 12*</u> *Let P, Q be two modules and x be a local variable in P. There is a* local-import data binding *triple (Q, x, P) if*

1. *$\exists$ a call $Q(\ldots, x, \ldots)$ in P such that the formal parameter of Q corresponding to x is a call-by-reference or call-by-value-result parameter and is defined within Q, and*

2. *x is referenced within P after the call.*

Synopsis: A local-import data binding triple (Q, x, P) represents the fact that module Q defines the value of the formal parameter a, and the corresponding actual parameter in a call to Q within P, is referenced within P after the call. It reflects the data relationship between modules P and Q due to the local variable-formal parameter bindings, and the direction of the information flow (from Q to P).

<u>*Example 5*</u> *Local-import data binding triple* (Q, x, P)

```
P()                     Q(a:[by reference,by value-result])
{ x:local               {
 Q(x)                     a<- // is defined
 <-x // is referenced
}                       }
```

The data bindings due to return value relationships are handled similar to local-import data bindings. First, one of several transformations are performed on the function invocation and invoked function definition as follows:

- Invoked function definition transformations:

1.
```
 C CODE                   TRANSFORMATION
Q(b,c)                   Q(a,b,c)
{                        {
 ...                      ...
 return a;                a<- // is defined
 ...                      ...
}                        }
```
2.
```
 C CODE                   TRANSFORMATION
Q(b,c)                   Q(<retval>,b,c)
{                        {
 ...                      ...
 return val;                <retval> <- val // is defined
 ...                      ...
}                        }
```

- Function invocation transformations:

1.
```
 C CODE                   TRANSFORMATION
P(...)                   P(...)
{x;                      {x;
 ...                      ...
 x = Q(y,z);              Q(x,y,z);
 <- x // is used          <- x // is used
 ...                      ...
}                        }
```
2.
```
 C code                   transformation
P(...)                   P(...)
{                        {x;
 ...                      ...
 exp(Q(y,z));             Q(x,y,z);
 ...                      exp(x);
                          ...
}                        }
```

Definition 13 *Let P, Q be two modules and P uses the return value from Q after a call $Q(...)$ in P. Perform one of two kind of transformations on the invoked module definition and the module invocation. Let a or $< retval >$ be the formal parameter resulting from the transformation on the invoked module definition. There is a* return-value data binding *triple (Q, a, P) or $(Q, < retval >, P)$ if*

1. *$\exists$ a call $Q(x)$ in P such that x is a transformation-generated local variable in P corresponding to the transformation-generated formal parameter a or $< retval >$ in Q that is defined within Q, and*
2. *x is referenced within P after the call.*

Synopsis: A return-value data binding triple (Q, a, P) or $(Q, < retval >, P)$ represents the fact that module Q defines a return-value, returns it to module P, and this return value is referenced within module P after the call. Either, the return-value is saved in a variable after the call for later reference, or it is directly referenced after the call. Notice that the data binding is expressed in terms of the return value variable as opposed to the local variable in the invoking module.

Definition 14 *Let P, Q be two modules and g be a global variable in P. There is a* global-export data binding *triple (P, g, Q) if*

1. *g is defined within P, before a call $Q(\ldots, g, \ldots)$ in P, and*
2. *the formal parameter of Q corresponding to g is referenced within Q.*

Synopsis: A global-export data binding triple (P, g, Q) is symmetric to the local-export data binding triple except that in the former the local variable x is replaced by a global variable g.

Example 6 *Global-export data binding triple (P, g, Q)*

```
P()                    Q(a:[by reference,by value-result, by value])
{                      {
 g<- // is defined       <-a // is referenced
 Q(g)
}                      }
```

Definition 15 *Let P, Q be two modules and g be a global variable in P. There is a* global-import data binding *triple (Q, g, P) if*

1. *$\exists$ a call $Q(\ldots, g, \ldots)$ in P such that the formal parameter of Q corresponding to g is a call-by-reference or call-by-value-result parameter and is defined within Q, and*

2. *g is referenced within P after the call.*

Synopsis: A global-import data binding triple (Q, g, P) is symmetric to the local-import data binding triple except that in the former the local variable x is replaced by a global variable g.

Example 7 *Global-import data binding triple (Q, g, P)*

```
P()                     Q(a:[by reference,by value-result])
{                       {
 Q(g)                     a<- // is defined
 <-g // is referenced
}                       }
```

Whenever it is not confusing, we will use the term *data binding* interchangeably with any of the forms of data bindings above.

Definition 16 *Let $Type(v)$ be the type of variable v.* Type size *of variable v, denoted $Tsize(v)$, is the amount of information carried by type $Type(v)$ of variable v.*

$Tsize(v)$ quantifies the information level associated with variable v due to the its type. In a conventional programming language, such as C or Ada, a primitive type is the least difficult to understand of its types. Since a pointer represents the address

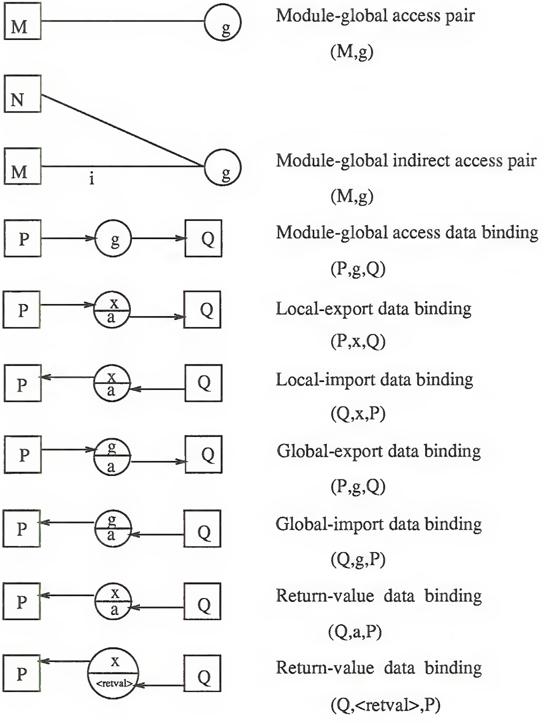

Figure 5.1. Schematic illustrations of access pairs and data bindings

Table 5.1. Type size associated with several types.

TYPE (Based on "C"/ "Ada")		TSIZE
primitive types		
	e.g.: int, char, float,	1
	void,boolean,enum	1
pointer		1
array		f(Tsize(array[i]), No. of elements, Control Info Size)
user-defined types; e.g:struct, union		Sum of the Tsizes of elements

of an object, we conclude that it is as easy to understand as primitive types. The user-defined types, such as structures and unions, are more complicated because their components' information level; thus, we assume that the difficulty of understanding a user-defined type is the added difficulty of understanding each of its components. Finally, an array has special mechanisms to manipulate its elements which adds to the difficulty of understanding the array.

Table 5.1 specifies the type size for the type found in two conventional programming languages. The higher the $Tsize$ value, the more information is carried by the type and it is more difficult to understand a variable of such type. For arrays, *control info size* refers to the complexity associated with the array control mechanisms of the particular language under analysis (e.g.: in C, type control size = 2: 1 unit to account for the index - offset - used in addressing the individual elements of the array and 1 unit to account for the storage of the address of the beginning of the array).

<u>*Definition 17*</u> *A* group *G is a pair of sets of modules and global variables.*

<u>*Definition 18*</u> *The* boundary *of group G is the set of data bindings (M_i, v, M_j) such that either $M_i \in G$ and $M_j \notin G$, or $M_i \notin G$ and $M_j \in G$.*

In the case of a module-global access data binding (M_i, g, M_j) in the boundary of group G, there are two associated access pairs in the boundary, namely (M_i, g) and (M_j, g).

Definition 19 *The* interior *set of group G is the set of data bindings (M_i, v, M_j) such that both $M_i \in G$ and $M_j \in G$.*

Let $|S|$ be the number of elements in set S.

5.3.2 Inter-group Complexity Factors

The *inter-group complexity* of group G measures the complexity in the interface between the group G and other groups of the system. The inter-group complexity of group G is characterized by the union of all the primitive measurements of inter-group complexity. The total inter-group complexity of a system is the average inter-group complexity of the groups in the system. Another interesting measure is to consider the maximum inter-group complexity of all the groups in the system.

5.3.2.1 Data-based primitive measurements

In this section, we present the primitive measurements of the inter-group complexity which are based on the data of the system.

- $f_1(G)$ is the set of global variables used by modules inside group G and also used by modules outside group G, or the set of direct interface variables, i.e.:

  $f_1(G) = \{\ g \mid \exists$ module-global access pairs (M, g) *and* (N, g) *such that* $M \in G$ *and* $N \notin G.\}$

  Assume that a global variable which is concurrently used by modules in two groups, increases the inter-group complexity of both groups.

We expect that the more global variables are shared between a group and other groups, the more this group is related to these other groups. The relationship is due to the potential of sharing the same data space among several groups.

Beladi's intercluster complexity [3] is proportional to N, the total number of nodes (where nodes are either modules or global variables) in a system. This complexity is given by $C_o = N * E_o$ where E_o is the number of "intercluster" edges. Thus, given the number of global variables N, we conclude that the number of global variables increases the inter-group complexity.

- $f_2(G)$ is the set of global variables outside of group G indirectly used by modules within group G, or the set of indirect interface variables, i.e.:

  $f_2(G) = \{\ g \mid \exists$ module-global indirect access pair (M, g) *such that* $M \in G$ *and* $g \notin G.\}$

  We expect that the more global variables which indirectly affect or are affected by modules in the group, the probability that the group is indirectly related to other groups increases. This relationship is due to the indirect knowledge about a global that the module must have.

  In this factor, Beladi's intercluster complexity is also used to verify our intuition. Since the intercluster complexity is proportional to N, the total number of nodes in a system, we conclude that the number of global variables increases the inter-group complexity.

- $f_3(G)$ is the boundary set of group G, i.e.:

  $f_3(G) = \{(M_i, v, M_j) \mid \exists$ *a data binding* (M_i, v, M_j) *in the boundary set of* $G.\}$

  The intercluster complexity is proportional to the number of "intercluster edges" [3]. Accordingly, the more intercluster edges there is, the more complex is the

interface of the system. In this factor, the data bindings across groups boundaries correspond to the intercluster edges. Hence, the complexity increases with the number of data bindings. A similar observation was made by Henry and Kafura [11] in the case of modularity metrics.

- $f_4(G)$ is the set of different variables transferring information between group G and other groups, or the set of variables in the boundary set, i.e.:

  $f_4(G) = \{v \mid \exists\ a\ data\ binding\ (M_i, v, M_j)\ in\ the\ boundary\ set\ of\ G.\}$

  Assume that variables which transfer information between the group and other groups, thus defining data bindings, relate the group to those other groups. Hence, the more variables relate a group to other groups, the inter-group complexity increases. Similarly, Beladi's intercluster complexity considers unique nodes, where a node is either a module or a global data in the system, as a factor which increases intercluster complexity. Factor f_4 considers unique occurrence of variables in the boundary set.

5.3.2.2 Type-based primitive measurements

In this section, we present the primitive measurements of the inter-group complexity which are based on the types of data in the system.

Assume that the information level of a variable is determined by its type size (Tsize) according to Definition 16.

- $f_5(G) =$ is the sum of type sizes of the global variables used by group G and also used by modules outside group G, or the sum of type sizes of direct interface variables, i.e.:

  $$f_5(G) = \sum_{g \in \{g \mid \exists\ module - global\ access\ pairs\ (M,g)\ and\ (N,g)\ such\ that\ M \in G\ and\ N \notin G.\}} Tsize(g)$$

We expect that the higher total type size of all global variables concurrently used by modules inside and outside the group, the probability that the group is related to those other groups increases and the inter-group complexity increases.

- $f_6(G) =$ is the sum of type sizes of global variables outside group G indirectly used by modules within group G, or the sum of type sizes of indirect interface variables, i.e.:

  $$f_6(G) = \sum_{g \in \{g \mid \exists\ module - global\ indirect\ access\ pair\ (M,g)\ such\ that\ M \in G\ and\ g \notin G.\}} Tsize(g)$$

  We expect that the higher total type size of the global variables outside the group indirectly used by modules inside the group, the probability that the group is related to those other groups increases and the inter-group complexity increases.

- Consider the set of data binding triples in the boundary set of a group. $f_7(G)$ is the sum of type sizes of boundary set variables.

  The sum of the type sizes of different variables passing data into group G is:

  $$f_{7.1}(G) = \sum_{v \in \{v \mid \exists\ data\ binding\ (M_i,v,M_j) \in\ boundary\ set\ of\ G\ \ni\ M_i \notin G\ and\ M_j \in G.\}} Tsize(v)$$

  The sum of the type sizes of different variables passing data out of group G is:

  $$f_{7.2}(G) = \sum_{v \in \{v \mid \exists\ data\ binding\ (M_i,v,M_j) \in\ boundary\ set\ of\ G\ \ni M_i\ \in G\ and\ M_j \notin G.\}} Tsize(v)$$

  Assume that variables which transfer information between the group and other groups, defining data bindings, contribute to the inter-group complexity of the group according to the variable's type size. We expect that the higher total type size of the variables passing data to or from the group, the inter-group complexity increases.

- $f_8(G)$ is the set of different types of variables passing data between group G and other groups, or the set of types of boundary set variables, i.e.:

  $f_8(G) = \{Type(v) \mid \exists\ data\ binding(M_i, v, M_j) \in\ boundary\ set\ of\ G.\}$

  Assume that each type of variable passing data to or from the group needs to be supported by the group's interface. Each type increases the inter-group complexity. Consequently, the more different types of variables are supported, the inter-group complexity increases. That is due to the different behaviors which need to be supported by the group's interface.

5.3.3 Intra-group Complexity Factors

The *intra-group complexity* of group G consists of the complexity of group G in a system partition. The intra-group complexity of group G is characterized by the union of all primitive measurements of intra-group complexity. The total intra-group complexity of a system is the sum of the intra-group complexity of all the system's groups.

Another measure related to these factors is the *intra-group strength*[1] which is a measure of the relationship between the modules in a group.

5.3.3.1 Data-based primitive measurements

In this section, we present the primitive measurements of the intra-group complexity which are based on the data of the system.

- $f_9(G)$ is the set of global variables used exclusively by modules in group G, or the set of direct internal variables, i.e.:

  $f_9(G) = \{\ g \mid \forall$ module-global access pair $(M_i, g) \ni\ M_i \in G.\}$

[1]This term was originally used by Myers [27].

- $f_{10}(G)$ is the set of global variables used indirectly only by modules in group G, or the set of indirect internal variables, i.e.:

  $f_{10}(G) = \{\ g \mid \forall$ module-global indirect access pair $(M_i, g) \ni\ M_i \in G.\}$

  Factors f_9 and f_{10} above consists of the global variables "exclusively used" by modules in the group.

  Assume that a global variable exclusively used by modules in the group weakens the intra-group strength and increases the intra-group complexity. According to [3], a global variable negatively affects the intracluster complexity by increasing its complexity value, given a set of edges. Consequently, a global variable increases the intra-group complexity. Also, we argue that a global variable negatively affects the intra-group strength since other groups may use this global variable thus reducing the functional relatedness of the group.

  We expect that the more global variables (exclusively used by modules in the group and thus not used by modules in other groups), the intra-group complexity increases. An extreme case occurs when the global variable becomes a *local variable* with respect to the group. Similarly, we expect that the strength between modules in the group weakens.

- Consider the set of data bindings triples in the interior set of a group. $f_{11}(G)$ is the interior set of group G, i.e.:

  $f_{11.1}(G)$ is the set of module-global access data bindings involving modules within group G, i.e.:

  $f_{11.1}(G) = \{(M_i, v, M_j) \mid \exists\ a$ module-global access data binding (M_i, v, M_j) *in the interior set of* $G.\}$

$f_{11.2}(G)$ is the set of local-export data bindings involving modules within group G, i.e.:

$f_{11.2}(G) = \{(M_i, v, M_j) \mid \exists\ a$ local-export data binding (M_i, v, M_j) *in the interior set of* $G.\}$

$f_{11.3}(G)$ is the set of local-import data bindings involving modules within group G, i.e.:

$f_{11.3}(G) = \{(M_i, v, M_j) \mid \exists\ a$ local-import data binding (M_i, v, M_j) *in the interior set of* $G.\}$

$f_{11.4}(G)$ is the set of global-export data bindings involving modules within group G, i.e.:

$f_{11.4}(G) = \{(M_i, v, M_j) \mid \exists\ a$ global-export data binding (M_i, v, M_j) *in the interior set of* $G.\}$

$f_{11.5}(G)$ is the set of global-import data bindings involving modules within group G, i.e.:

$f_{11.5}(G) = \{(M_i, v, M_j) \mid \exists\ a$ global-import data binding (M_i, v, M_j) *in the interior set of* $G.\}$

$f_{11.6}(G)$ is the set of return-value data bindings involving modules within group G, i.e.:

$f_{11.6}(G) = \{(M_i, v, M_j) \mid \exists\ a$ return-value data binding (M_i, v, M_j) *in the interior set of* $G.\}$

According to Beladi and Evangelisti [3], the intracluster complexity of a single cluster j is $C_j = n_j * e_j$ where n_j is the number of nodes in the cluster and e_j the number of pairwise edges, that is, "connections between the same elements." Given that edges correspond to data bindings in our model, we conclude that

the more data bindings between the modules in a group, the higher the intra-group complexity of the group.

Furthermore, the intra-group strength increases with the number of data bindings since the more "channels" of data passing between modules in the group, the higher the functional relatedness of the group.

It appears as if factors f_{11} conflict with factors f_9 and f_{10}; however, data bindings define relations between modules which theoretically speaking will increase the strength. However, the fact that the data bindings are sometimes due to global variables involves a risk in that other groups in the system may access these global variables which weakens the strength of the group.

- This factor applies to groups derived from using the globals-based analysis (Algorithm 1): $f_{12}(G)$ is the percentage of modules in group G which directly use all global variables in group G, i.e.:

  $f_{12}(G) = |\{M \mid \exists \text{ module-global access pair } (M, g_i)\ \forall\ \textit{global variable } g_i \in G \textit{ and } M \in G.\}|/|\{M \mid M \in G\}|$

  Assume that a global variable used by all modules in the group increases the intra-group strength. The higher the ratio of the number of modules which directly use all global variables in the group to the total number of modules, the intra-group strength increases.

5.3.3.2 Type-based primitive measurements

In this section, we present the primitive measurements of the intra-group complexity which are based on the types of data in the system.

Assume that the information level of a variable is determined by its type size (Tsize) according to Definition 16.

- Consider the set of data bindings triples in the interior set of group G.

  Also, consider only the type of different variables since we measure the complexity due to the kind of information passed between modules of the group, as opposed to the volume of this information.

  $f_{13}(G)$ is the sum of the type sizes of different variables passing data between modules M_i and M_j in group G:

  $$f_{13}(G) = \sum_{v \in \{v \mid \exists \; data\ binding\ (M_i, v, M_j) \in \; interior\ set\ of\ G.\}} Tsize(v)$$

  We expect that the higher total type size of all the variables used by modules in the group, the intra-group complexity increases and the intra-group strength decreases.

- $f_{14}(G)$ is the set of types of different variables passing data between modules M_i and M_j in group G, or the set of types of interior set variables, i.e.:

  $$f_{14}(G) = \; \{Type(v) \mid \exists \; data\ binding\ (M_i, v, M_j) \in \; the\ interior\ set\ of\ G.\}$$

  Assume that each type of variable passing data among the modules in the group needs to be supported by the module. Each type increases the intra-group complexity. Therefore, the more different types of variables are supported, the intra-group complexity increases. Also, less commonality is observed in the group and the intra-group strength decreases due to the reduced functional relatedness of the group.

- The following factors only apply to groups derived from using the types-based analysis (Algorithm 2):

<u>*Definition 20*</u> *The* Base Type *of group G, denoted $Btype(G)$, is the type used as the grouping criteria during the "grouping" step of the Types-based Object Finder, Algorithm 2.*

<u>*Definition 21*</u> *Module M* manipulates *type t, denoted by the* manipulation pair *$[M, t]$, if t is the type of a formal parameter of M, or t is the type of the return value of M, or $t = Type(g)$ such that $\exists$ access pair (M, g). This is, t is one of the types that module M may manipulate.*

<u>*Definition 22*</u> *A* grouping manipulation pair *in group G is a manipulation pair $[M, t]$ such that $M \in G$ and $t \equiv Btype(G)$.*

There exists a grouping manipulation pair for a module M, $M \in G$, and type t, denoted $[M, t]$, iff any of module M types of formal parameters or return value, t, is equivalent[2] to the base type of group G, $Btype(G)$; i.e.: $t \equiv Btype(G)$. Also, there exists a grouping manipulation pair for a module M, $M \in G$, and type of variable v, $Type(v)$, denoted $[M, Type(v)]$, iff there exists a module-global access pair (M, v) and the type of variable v, $Type(v)$, is equivalent to the base type of group G; i.e.: $Type(v) \equiv Btype(G)$.

$f_{15}(G)$ is the set of grouping manipulation pairs $[M, t]$ such that $M \in G$, i.e.:

$f_{15}(G) = \{[M, t]$ *is a grouping manipulation pair* $\mid M \in G.\}$

We expect that the more grouping manipulation pairs, the intra-group strength increases. The evidence indicates that this factor does not affect the intra-group complexity.

[2]Equivalent types are defined in Algorithm 2 in Chapter 3.

- <u>*Definition 23*</u> *A* degrouping manipulation pair *in group G is a manipulation pair $[M, t]$ such that $M \in G$ and $t \ll Btype(G)$.*

  There exists a degrouping manipulation pair for a module M, $M \in G$, and type t, denoted $[M, t]$, iff any of module M types t of formal parameters or return value is a "part of" the base type of group G or "contains" type that is the base type of group G, $Btype(G)$; i.e.: $t \ll Btype(G)$ or $t \gg Btype(G)$. Also, there exists a degrouping manipulation pair for a module M, $M \in G$, and type of variable v, $Type(v)$, denoted $[M, Type(v)]$, iff there exists a module-global access pair (M, v) and the type of variable v, $Type(v)$, is a "part of" the base type of group G or "contains" type that is the base type of group G, $Btype(G)$; i.e.: $Type(v) \ll Btype(G)$ or $Type(v) \gg Btype(G)$.

  $f_{16}(G)$ is the set of degrouping manipulation pairs $[M, t]$ such that $M \in G$, i.e.:

  $f_{16}(G) = \{[M, t]$ *is a degrouping manipulation pair* $\mid M \in G.\}$

  We expect that the more degrouping manipulation pairs, the intra-group strength decreases. Once more, this factor does not affect the intra-group complexity.

- $f_{17}(G) =$ the ratio of number of grouping manipulation pairs to the number of degrouping manipulation pairs.

  $f_{17}(G) = |f_{15}(G)| : |f_{16}(G)|$

  Given factors f_{15} and f_{16} above, we expect that the higher the ratio of grouping manipulation pairs to degrouping manipulation pairs, the intra-group strength increases.

  In the case that types t and $Btype(G)$ are not explicitly related (e.g., types are equivalent or there is a "part of" or "contains" relationship between the types),

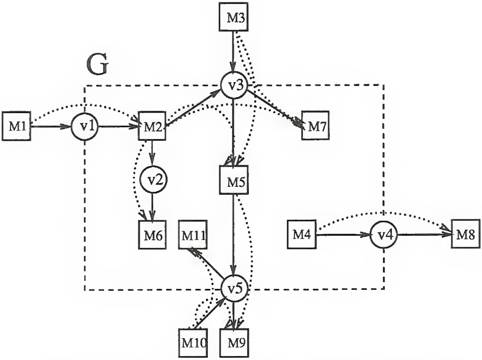

Figure 5.2. Example of the primitive complexity metrics factors

the corresponding manipulation pairs of the form $[M, t]$ are not considered in factors f_{15}, f_{16}, and f_{17}.

5.3.4 Example of the Factors

This section presents an example of the primitive complexity metrics factors. Figure 5.2 consists of a set of several routines, global variables, and a group.

Examples of the complexity metrics factors, in Figure 5.2, related to group G are:

1. *Inter-group Complexity*

   - f_3 Boundary set:

     $f_3(G) = \{(M_1, v_1, M_2), (M_3, v_3, M_5), (M_3, v_3, M_7), (M_4, v_4, M_8),$
     $(M_5, v_5, M_9), (M_{10}, v_5, M_{11}), (M_{10}, v_5, M_{11})\}$

- f_4 Variables in boundary set: $f_4(G) = \{v_1, v_3, v_4, v_5\}$
- f_7 Sum of type sizes of different variables in boundary set: $f_7(G) = Tsize(v_1) + Tsize(v_3) + Tsize(v_4) + Tsize(v_5)$

2. *Intra-group Complexity*

- f_{11} Interior set:
  $f_{11}(G) = \{(M_2, v_2, M_6), (M_2, v_3, M_7), (M_2, v_3, M_5), (M_5, v_5, M_{11})\}$
- f_{13} Sum of type sizes of different variables in interior set: $f_{13}(G) = Tsize(v_2) + Tsize(v_3) + Tsize(v_5)$

5.3.5 Validation of the Factors

This section presents the validation of the complexity metrics factors of the previous sections. The validation approach consists of proving that the validation hypothesis holds. This approach is described in Figure 5.5. This section also illustrates the sensitivity of the factors to a particular modification of a program. More experimentation is needed to further validate these factors with respects to other kind of changes applied to programs. Other interesting modifications to programs include a functionally equivalent program with several functions merged into a single function or several abstract data types of the original program replaced with equivalent data structures.

The example program consists of a simple recursive descent expression parser implemented in "C". First, we present the complexity metrics computed on a version of this program which consists of two source files, eighteen functions, and four global variables. Second, we present the metrics computed on another version of this program with the same functionality except that most function parameters are converted into global variables; this version consists of the same number of source files as

Table 5.2. Type size associated with variables of different types in the original version of the recursive descent expression parser.

Variable	Type	TSIZE
prog	char*	1
tok_type	char	1
token	char[80]	$1 * 80 + 2 = 82$
vars	float[26]	$1 * 26 + 2 = 28$
~answer~	float	1
~op~	register char	1
~result~	float*	1
~c~	char	1
~hold~	float	1

the previous version but with a total of nineteen functions and six global variables. Third, we explain the effect that the structure of each version of the program had on the primitive complexity metrics factors and illustrate the sensitivity of the metrics to this class of changes in the structure of a program.

In the first version of the program, named the original version, the top-down analysis approach was used with both globals-based and types-based analyses; we choose to ignore C primitive types during the types-based analysis. The identified objects are shown in Figure 5.3. Some modifications of the candidate objects were performed to obtain completely disjoint candidate objects in terms of their routines. These modifications consisted of removing common components (routines) from object `Gtok_type#2` to a single candidate object. The resulting objects after the modifications are shown in Figure 5.3.

Table 5.2 illustrates the types sizes corresponding to the types of variables in the original version of the example. The primitive metrics factors were computed for inter-group complexity, intra-group complexity, and intra-group strength.

```
Object z#objTchar*#3 is {
   ~~char*~76                              :(T)char*
   Float_Parsing~___II~iswhite~181         :(R)iswhite
   Float_Parsing~___II~isdelim~175         :(R)isdelim
   Float_Parsing~___II~is_in~172           :(R)is_in
}
Object z#objTfloat*#4 is {
   ~~float*~24                             :(T)float*
   Float_Parsing~___II~primitive~240       :(R)primitive
   Float_Parsing~___II~level6~198          :(R)level6
   Float_Parsing~___II~level5~196          :(R)level5
   Float_Parsing~___II~level4~194          :(R)level4
   Float_Parsing~___II~level3~192          :(R)level3
   Float_Parsing~___II~level2~190          :(R)level2
   Float_Parsing~___II~level1~188          :(R)level1
   Float_Parsing~___II~get_exp~143         :(R)get_exp
}
Object z#objT+undetermined-type#5 is {
   ~~float*~24                             :(T)float*
   ~~char*~76                              :(T)char*
   Float_Parsing~___II~unary~371           :(R)unary
   Float_Parsing~___II~find_var~100        :(R)find_var
   Float_Parsing~___II~arith~21            :(R)arith
}
Object z#objGtok_type#2 is {
   main~___II~prog~243                     :(G)prog
   Float_Parsing~___II~vars~374            :(G)vars
   Float_Parsing~___II~token~357           :(G)token
   Float_Parsing~___II~tok_type~356        :(G)tok_type
   main~___II~main~206                     :(R)main
   Float_Parsing~___II~putback~244         :(R)putback
   Float_Parsing~___II~get_token~145       :(R)get_token
   Float_Parsing~___II~find_var~100        :(R)find_var
}
```

Figure 5.3. Identified objects in original version of recursive descent expression parser

Table 5.3. Type size associated with variables of different types in version 1 of the recursive descent expression parser.

Variable	Type	TSIZE
`prog`	`char *`	1
`tok_type`	`char`	1
`token`	`char[80]`	$1 * 80 + 2 = 82$
`vars`	`float[26]`	$1 * 26 + 2 = 28$
`answer`	`float[1048]`	$1 * 1048 + 2 = 1050$
`result`	`float *`	1
`~op~`	`register char`	1
`~c~`	`char`	1
`~hold~`	`float`	1

For the other version of the program, named version 1, the top-down analysis approach was used with both globals-based and types-based analyses; we choose to ignore C primitive types during the types-based analysis. The identified objects are shown in Figure 5.4. Some modifications of the candidate objects were performed to obtain completely disjoint candidate objects, specifically routine `find_var` was removed from object `T+undetermined-type#4`. The resulting objects after the modifications are shown in Figure 5.4.

Table 5.3 illustrates the types sizes used for the metrics computation. In this version of the program, as well as the rest of the examples in this thesis, whenever possible, the name of an identifier denotes the identifier, instead of using the unique ID which is automatically generated by the object finder tool of Chapter 7. See Section 8.1 for a complete explanation of unique ID in the internal representation of the program used by the object finder tool.

The primitive metrics factors were computed for inter-group complexity, intra-group complexity, and intra-group strength.

```
Object z#objTchar*#3 is {
   ~~char*~77                              :(T)char*
   Float_Parsing~___II~iswhite~177         :(R)iswhite
   Float_Parsing~___II~isdelim~171         :(R)isdelim
   Float_Parsing~___II~is_in~168           :(R)is_in
}
Object z#objT+undetermined-type#4 is {
   ~~float*~25                             :(T)float*
   ~~char*~77                              :(T)char*
   Float_Parsing~___II~unary~362           :(R)unary
   Float_Parsing~___II~arith~22            :(R)arith
}
Object z#objGanswer#2 is {
   main~___II~prog~232                     :(G)prog
   Float_Parsing~___II~vars~365            :(G)vars
   Float_Parsing~___II~token~348           :(G)token
   Float_Parsing~___II~tok_type~347        :(G)tok_type
   Float_Parsing~___II~result~309          :(G)result
   Float_Parsing~___II~answer~20           :(G)answer
   main~___II~main~196                     :(R)main
   Float_Parsing~___II~result_inc~310      :(R)result_inc
   Float_Parsing~___II~putback~233         :(R)putback
   Float_Parsing~___II~primitive~230       :(R)primitive
   Float_Parsing~___II~level6~189          :(R)level6
   Float_Parsing~___II~level5~188          :(R)level5
   Float_Parsing~___II~level4~187          :(R)level4
   Float_Parsing~___II~level3~186          :(R)level3
   Float_Parsing~___II~level2~185          :(R)level2
   Float_Parsing~___II~level1~184          :(R)level1
   Float_Parsing~___II~get_token~145       :(R)get_token
   Float_Parsing~___II~get_exp~144         :(R)get_exp
   Float_Parsing~___II~find_var~101        :(R)find_var
}
```

Figure 5.4. Identified objects in version 1 of recursive descent expression parser

The approach to validate the complexity metrics consists of proving that the following hypothesis hold. The hypothesis states that the primitive complexity metrics factors are sensitive to changes in the complexity caused by different partitionings. Different partitionings are the result of (1) using different partitionings approaches under the same connectivity, or (2) totally different connectivities. The connectivity of a program is the set of relationships between components of the program defined by the data bindings among components due to global variables and calling sequences. We use the second situation to prove that the hypothesis holds.

The methodology to prove the hypothesis is to use a case study program to show that the primitive metrics reflect different complexity measures in two versions of the program which are functionally equivalent and have different connectivities as a result of having different structures. An schematic description of this validation methodology is given in Figure 5.5. From Figure 5.5, the validation of the metrics factors consists of proving the following hypothesis: *If* Co_1 *is less complex than* Co_2, *then* $Me_1 < Me_2$, where Co_1 is the expected complexity of the original version, Co_2 is the expected complexity of version 1, Me_1 is the measured primitive metrics factors values of the original version, and Me_2 is the measured primitive metrics factors values of version 1.

For the proof, we use the program above with two versions: the original version and version 1. We made version 1 to be more complex in terms of the connectivity of the program, since a program with greater connectivity is expected to be more complex given that all other factors remain the same. Version 1 is made more complex by replacing most formal parameters with global variables. Thus, the number of expected relations between components of the program will increase.

The expected complexity of each of the two versions is defined as the complexity resulting from the connectivies between components of a system. We expect that the

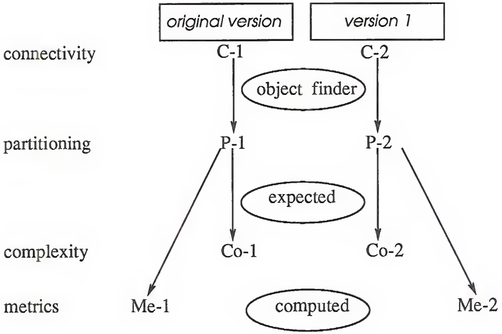

Figure 5.5. Validation of the primitive complexity metrics factors

groups in the partitioning obtained from version 1 are more intra-group complex than the groups in the partitioning obtained from the original version. This is the case since each group in version 1 partitionings is expected to be highly cohesive by the added global variables in version 1. We also expect that the groups in version 1 are less inter-group complex because the groups in version 1 partitioning are expected to be loosely connected by the global variables in version 1.

Each version of the program exhibits different connectivity between the program components. Hence, the object finder partitionings for each version of the program were different. In addition, the objective of the study that the program versions were functionally equivalent but structurally different was met.

The validation results for each complexity metric factor are summarized next. First, the results about the complexity metrics factors that measure inter-group complexity.

f_1 Set of direct interface variables   The output for the two versions of the program indicated that the original version presented higher inter-group complexity than version 1. This observation shows the effect of different partitionings on the metrics. In addition, as expected, there is higher interactions between components in the original version than between components in version 1.

f_2 Set of indirect interface variables   No relevant results were obtained for this factor.

f_3 Boundary set   The output for the two versions of the program indicated that the original version presented slightly higher inter-group complexity than version 1. This observation is in accordance with the expected variation in the metrics.

f_4 Variables in boundary set The output for the two versions of the program indicated that the two versions presented similar inter-group complexity.

f_5 Sum of type sizes of direct interface variables The output for the two versions of the program indicated that the original version presented higher inter-group complexity than version 1. This is the case since version 1 partitioning reduces the number of global variables transferring information between components.

f_6 Sum of type sizes of indirect interface variables No relevant results were obtained for this factor.

f_7 Sum of type sizes of boundary set variables The output for the two versions of the program indicated that the original version presented lower inter-group complexity than version 1. Version 1 added some global variables to the program. They influenced the sizes observed in the new program. However, the increased value of the type size is related to the number of relations (data bindings) between components. This number was lower in version 1 than in the original version.

f_8 Set of types of boundary set variables The output for the two versions of the program indicated that the original version presented higher inter-group complexity than version 1. This is the case since version 1 partitioning reduces the number of variables transferring information between components.

In conclusion, the inter-group complexity in version 1 is lower than the one in the original version. This coincides with our expectations regarding the effect of the changes and the resulting partitioning on the primitive complexity metrics factors. That is to say, the partitioning resulting from version 1 agglomerates most relations between components inside the groups which reduces the inter-group complexity.

f_9 Set of direct internal variables The output for the two versions of the program indicated that version 1 presented higher inter-group complexity than the original version. This coincides with our expectations that the intra-group complexity will increase in version 1 since more global variables are used.

f_{10} Set of indirect internal variables The output for the two versions of the program indicated that the two versions presented similar intra-group complexity. One reason for such situation is the fact that the code related to factor f_{10} is not modified between versions.

f_{11} Interior set The output for the two versions of the program indicated that the original version presented lower intra-group complexity than version 1. This coincide with our expectation of the intra-group complexity will increase in version 1 since more global variables are used.

f_{12} Percentage of modules accessing all global variables No relevant results were obtained for this factor.

f_{13} Sum of type sizes of interior set variables The output for the two versions of the program indicated that the original version presented lower intra-group complexity than version 1. This demonstrates that factor f_{13} increases with a higher intra-group complexity partitioning, such as the one in version 1 of the program.

f_{14} Set of types of interior set variables The output for the two versions of the program indicated that the original version presented lower intra-group complexity than version 1. This coincides with our expectations that the intra-group complexity will increase in version 1 since more relations occur.

In conclusion, the intra-group complexity in version 1 is higher than the intra-group complexity in the original version. This coincides with our expectation regarding the effect of version 1 partitioning on the metrics. That is to say, the partitioning resulting from version 1 agglomerates most relations between components inside the groups which increases the intra-group complexity.

f_{15} Set of grouping manipulation pairs   The output for the two versions of the program indicated that the original version presented higher intra-group strength than version 1. This coincides with our expectation that the partitioning of version 1 decreases the strength because parameters have been replaced with global variables and, as previously indicated, the addition of global variables reduces the strength of individual components.

f_{16} Set of degrouping manipulation pairs   The output for the two versions of the program indicated that the original version presented higher intra-group strength than version 1. This coincides with our expectations that the partitioning resulting from version 1 decreases the strength.

f_{17} Ratio of grouping to degrouping manipulation pairs   The output for the two versions of the program indicated that the original version presented higher intra-group strength than version 1. This coincides with our expectations that the partitioning resulting from version 1 decreases the strength.

In conclusion, the intra-group strength in the original version of the program is higher than the one in version 1. This coincides with our expectations regarding the effect of version 1 partitioning on the primitive metrics factors. That is to say, version 1 reduces the strength of the resulting components due to the elimination of the parameters from the routines interface which were replaced with global variables.

The validation of these primitive metrics factors shows a correlation between the primitive metrics factors and the expected complexity associated with a partitioning of a system. We conclude that the complexity of a partitioning is effectively measured by these primitive complexity metrics factors. Different complexities will result in different values of the primitive metrics factors. The future research consists of decribing the relationships between complexity metrics factors in terms of a formula.

5.4 The Test Cases: Identified Objects, Clusters and Groups

The following sections present the partitionings identified in the three test case programs using two modularization techniques. The partitionings consists of the identified objects using the top-down analysis method of the object finder and the clusters identified using Hutchens and Basili hierarchical clustering technique [14]. A generalized view of both kind of partitionings consists of a set of *groups*, from Definition 17; a group usually corresponds to an identified object in the object finder and to a cluster in hierarchical clustering. In the following sections, we explain how the groups based on identified objects and on cluster were specified for each test case.

5.4.1 Test Case 1: Name Cross-reference Program

The first test case consists of the name cross-reference program from Section 8.1. The statistics of this example are given in Table 5.4. This section presents the identified objects by the object finder using the top-down analysis method. In addition, we present the clusters identified in this program using Basili's hierarchical clustering technique [14].

The identified objects, obtained by the object finder during the top-down analysis method, are shown in Figure 8.1. The groups corresponding to the identified objects in Figure 8.1 are derived after the user performs some modifications on the

Table 5.4. Statistics of the test case programs.

Program name	Lines of code	Global variables	Functions	Types
Name cross-reference	282	1	10	10
Algebraic expression evaluation	1324	10	50	14
Table management	1,900	8	50	4

identified objects. The purpose of the user's modifications is to eliminate the commonality between objects by removing common components between objects, as explained in Section 8.1, using `xobject`, which results in the objects of Figure 8.3; these disjoint objects were used to derive the groups. The groups are shown in Figure 5.6. Since these groups are based on objects after the modifications, we name a group by using the name of the object that correspond to the group.

The clusters of this test case were identified by a clustering tool called `basili` [21] that implements Basili's hierarchical clustering technique. The identified clusters in the cross-reference program are shown in Figure 5.7.

Figure 5.7 showed the clusters defined on this example using Basili's clustering techniques. The groups corresponding to these clusters are shown in Figure 5.8. In this case, a group consists of one or more clusters of routines which maintain the same levels of coupling and strength [14] of the cluster identified with the hierarchical clustering technique. The groups are constructed as follows. Initially, groups are defined based on the top level of corresponding clusters. In this case, the naming of groups is arbitrary and only serves to distinctly identify each group. Then, the routines which did not cluster were considered to be groups of size one. An alternative approach, during this last step, is to group together the unclustered routines.

```
Group z#objTLINEPTR#3 is {
    ~~LINEPTR~16 :(T)LINEPTR
    xref_tab~___II~make_linenode~138        :(R)make_linenode
}
Group z#objTWTPTR*#4 is {
    ~~WTPTR*~104 :(T)WTPTR*
    xref_tab~___II~make_wordnode~140        :(R)make_wordnode
    xref_tab~___II~init_tab~102             :(R)init_tab
    xref_tab~___II~addword~64               :(R)addword
    xref_tab~___II~add_lineno~61            :(R)add_lineno
    xref_out~___II~writewords~181           :(R)writewords
}
Group z#objTchar*#5 is {
    ~~char*~56                              :(T)char*
    xref_tab~___II~strsave~164              :(R)strsave
}
Group z#objTint*#6 is {
    ~~int*~69 :(T)int*
    xref~___II~main~137 :(R)main
}
Group z#objGlineno#2 is {
    xref_in~___II~lineno~130                :(G)lineno
    xref_in~___II~getword~98                :(R)getword
    xref_in~___II~getachar~93               :(R)getachar
}
```

Figure 5.6. Groups based on objects identified in name cross-reference program

```
Final dendrogram:
 -- Cluster No.1 --
(100 (50 add_lineno addword)
     make_linenode
     (50 make_wordnode strsave) )
 -- Cluster No.2 --
(100 getachar getword)
```

Figure 5.7. Clusters found in the name cross-reference program by `basili`

```
Group I is {
    xref_tab~___II~add_lineno~61      :(R)add_lineno
    xref_tab~___II~addword~64         :(R)addword
    xref_tab~___II~make_wordnode~140  :(R)make_wordnode
    xref_tab~___II~strsave~164        :(R)strsave
    xref_tab~___II~make_linenode~138  :(R)make_linenode
}
Group II is {
    xref_in~___II~getword~98          :(R)getword
    xref_in~___II~getachar~93         :(R)getachar
}
Group III is {
    xref_tab~___II~init_tab~102       :(R)init_tab
}
Group IV is {
    xref_out~___II~writewords~181     :(R)writewords
}
Group V is {
    xref~___II~main~137               :(R)main
}
```

Figure 5.8. Groups based on clusters found in name cross-reference program

5.4.2 Test Case 2: Algebraic Expression Evaluation Program

The second test case consists of the simple algebraic expression evaluation program from Section 8.2. The statistics of this program are listed in Table 5.4. This section presents the identified objects by the object finder during the top-down analysis method. In addition, we present the clusters identified in this program using Hutchens and Basili hierarchical clustering technique [14].

The identified objects, obtained by the object finder during the top-down analysis method, are shown in Figures 8.4 and 8.5. The groups corresponding to these identified objects are derived after the user performs modifications on the identified objects. Similarly to Section 5.4.1, we name a group based on the name of the object that corresponds to the group. The groups are shown in Figures 5.9 and 5.10.

```
Group z#objTNODE*#6 is {
   ~~NODE*~15                                   :(T)NODE*
   function~___II~Variable_eval~140             :(R)Variable_eval
   function~___II~Plus_eval~104                 :(R)Plus_eval
   function~___II~Node_eval~101                 :(R)Node_eval
   function~___II~Multiply_eval~96              :(R)Multiply_eval
   function~___II~Minus_eval~94                 :(R)Minus_eval
   function~___II~Function_precedence~82        :(R)Function_precedence
   function~___II~Function_delete_function      :(R)Function_delete_function
   function~___II~Function_checkandadd~68       :(R)Function_checkandadd
   function~___II~Function_build_tree~66        :(R)Function_build_tree
   function~___II~Function_add_operator~62      :(R)Function_add_operator
   function~___II~Function_add_operands~60      :(R)Function_add_operands
   function~___II~Echo_tree0~29                 :(R)Echo_tree0
   function~___II~Divide_eval~22                :(R)Divide_eval
   function~___II~Construct_function~16         :(R)Construct_function
   function~___II~Constant_eval~12              :(R)Constant_eval
   function~___II~Function_parenthesis~80       :(R)Function_parenthesis
   function~___II~Function_ovldop~73            :(R)Function_ovldop
   function~___II~total_paren~292               :(G)total_paren
   function~___II~root~267                      :(G)root
   function~___II~queue~263                     :(G)queue
   function~___II~last~216                      :(G)last
   function~___II~first~193                     :(G)first
}
Group z#objTchar*#7 is {
   ~~char*~18                                   :(T)char*
   state~___II~State9_transition~128            :(R)State9_transition
   state~___II~State7_transition~126            :(R)State7_transition
   state~___II~State6_transition~124            :(R)State6_transition
   state~___II~State5_transition~122            :(R)State5_transition
   state~___II~State4_transition~120            :(R)State4_transition
   state~___II~State3_transition~118            :(R)State3_transition
   state~___II~State2_transition~116            :(R)State2_transition
   state~___II~State1_transition~114            :(R)State1_transition
   state~___II~State0_transition~112            :(R)State0_transition
   function~___II~Echo_tree~27                  :(R)Echo_tree
}
```

Figure 5.9. Groups based on types-based objects identified in algebraic expression evaluation program

```
Group z#objTfloat#8 is {
   ~~float~5                              :(T)float
   function~___II~F6~52                   :(R)F6
   function~___II~F5~46                   :(R)F5
   function~___II~F4~41                   :(R)F4
   function~___II~F3~37                   :(R)F3
   function~___II~F2~34                   :(R)F2
   function~___II~F1~32                   :(R)F1
}
Group z#objTint#9 is {
   ~~int~4                                :(T)int
   exprtst~___II~main~217                 :(R)main
}
Group z#objTvoid#10 is {
   ~~void~2                               :(T)void
   function~___II~Destruct_function~21    :(R)Destruct_function
}
```

Figure 5.9 – – continued

Similarly to Section 5.4.1, the clusters of this test case were identified by a clustering tool called `basili` [21] based on Basili's hierarchical clustering technique. The identified clusters in the expression evaluation program are shown in Figure 5.11.

Figure 5.11 showed the clusters defined in this example using Basili's clustering techniques. The groups corresponding to these clusters are shown in Figure 5.12. In this test case, a group consists of one or more clusters and sub-clusters of routines which results in the same degree of coupling and strength [14] of the cluster identified by the hierarchical clustering technique. The groups are constructed as follows. First, a group corresponds to the set of routines which form sub-clusters with the lowest coupling and highest cohesion between the routines according to Hutchens and Basili [14] definition of strength and coupling. Next, the routines which did not cluster were grouped into a single group, named group `V`.

```
Group z#objGtable_index#2 is {
   symbol~___II~table~284                      :(G)table
   symbol~___II~table_index~286                :(G)table_index
   symbol~___II~Symbol_table_get_value~138 :(R)Symbol_table_get_value
   symbol~___II~Symbol_table_get_index~136 :(R)Symbol_table_get_index
   symbol~___II~Symbol_table_clear~135      :(R)Symbol_table_clear
   symbol~___II~Symbol_table_add_variable~ :(R)Symbol_table_add_variable
   symbol~___II~Symbol_table_add_value~130 :(R)Symbol_table_add_value
   symbol~___II~Echo_symbol_table~26        :(R)Echo_symbol_table
   symbol~___II~Construct_symbol_table~20  :(R)Construct_symbol_table
}
Group z#objGparen_count#3 is {
   state~___II~paren_count~254              :(G)paren_count
   state~___II~State0_inc_paren~111         :(R)State0_inc_paren
   state~___II~State0_get_paren~110         :(R)State0_get_paren
   state~___II~State0_dec_paren~108         :(R)State0_dec_paren
}
Group z#objGexpression#4 is {
   state~___II~index~213                    :(G)index
   state~___II~expression~178               :(G)expression
   state~___II~State0_get_char~109          :(R)State0_get_char
   state~___II~Construct_state0~19          :(R)Construct_state0
   function~___II~Function_valid~85         :(R)Function_valid
}
```

Figure 5.10. Groups based on globals-based objects identified in algebraic expression evaluation program

```
Final dendrogram:
 -- Cluster No.1 --
(9 Function_ovldop
   Symbol_table_add_value)

 -- Cluster No.2 --
(100 (66 Construct_function
         (62 Function_add_operands
             (44 Function_build_tree
                 Symbol_table_add_variable)
             Function_checkandadd) )
     Echo_queue Echo_tree0
     Function_add_operator
     Function_delete_function main)
```

Figure 5.11. Clusters found in algebraic expression evaluation program

```
Group I is {
   function~___II~Function_ovldop~73       :(R)Function_ovldop
   symbol~___II~Symbol_table_add_value~130 :(R)Symbol_table_add_value
}
Group II is {
   function~___II~Function_add_operator~62 :(R)Function_add_operator
   function~___II~Function_delete_function :(R)Function_delete_function
   exprtst~___II~main~217                  :(R)main
}
Group III is {
   function~___II~Construct_function~16    :(R)Construct_function
   function~___II~Function_add_operands~60 :(R)Function_add_operands
   function~___II~Function_checkandadd~68  :(R)Function_checkandadd
}
Group IV is {
   function~___II~Function_build_tree~66   :(R)Function_build_tree
   symbol~___II~Symbol_table_add_variable~ :(R)Symbol_table_add_variable
}
Group V is {
   function~___II~Variable_eval~140        :(R)Variable_eval
   function~___II~Plus_eval~104            :(R)Plus_eval
   function~___II~Node_eval~101            :(R)Node_eval
   function~___II~Multiply_eval~96         :(R)Multiply_eval
   function~___II~Minus_eval~94            :(R)Minus_eval
   function~___II~Function_precedence~82   :(R)Function_precedence
   function~___II~Echo_tree0~29            :(R)Echo_tree0
   function~___II~Divide_eval~22           :(R)Divide_eval
   function~___II~Constant_eval~12         :(R)Constant_eval
   function~___II~Function_parenthesis~80  :(R)Function_parenthesis
   state~___II~State9_transition~128       :(R)State9_transition
   state~___II~State7_transition~126       :(R)State7_transition
   state~___II~State6_transition~124       :(R)State6_transition
   state~___II~State5_transition~122       :(R)State5_transition
   state~___II~State4_transition~120       :(R)State4_transition
   state~___II~State3_transition~118       :(R)State3_transition
   state~___II~State2_transition~116       :(R)State2_transition
   state~___II~State1_transition~114       :(R)State1_transition
   state~___II~State0_transition~112       :(R)State0_transition
   function~___II~Echo_tree~27             :(R)Echo_tree
   function~___II~F6~52                    :(R)F6
   function~___II~F5~46                    :(R)F5
   function~___II~F4~41                    :(R)F4
   function~___II~F3~37                    :(R)F3
   function~___II~F2~34                    :(R)F2
   function~___II~F1~32                    :(R)F1
}
```

Figure 5.12. Groups based on clusters found in algebraic expression evaluation program

```
Group V (cont.) is {
   function~___II~Destruct_function~21      :(R)Destruct_function
   symbol~___II~Symbol_table_get_value~138 :(R)Symbol_table_get_value
   symbol~___II~Symbol_table_get_index~136 :(R)Symbol_table_get_index
   symbol~___II~Symbol_table_clear~135     :(R)Symbol_table_clear
   symbol~___II~Echo_symbol_table~26       :(R)Echo_symbol_table
   symbol~___II~Construct_symbol_table~20  :(R)Construct_symbol_table
   state~___II~State0_inc_paren~111        :(R)State0_inc_paren
   state~___II~State0_get_paren~110        :(R)State0_get_paren
   state~___II~State0_dec_paren~108        :(R)State0_dec_paren
   state~___II~State0_get_char~109         :(R)State0_get_char
   state~___II~Construct_state0~19         :(R)Construct_state0
   function~___II~Function_valid~85        :(R)Function_valid
}
```

Figure 5.12 – – continued

5.4.3 Test Case 3: Symbol Table Management for Acx

The third test case consists of the symbol table management module of an ANSI "C" parser tool. The statistics of this program are listed in Table 5.4. This section presents the identified objects by the object finder during the top-down analysis method. In addition, we present the clusters identified in this program using Hutchens and Basili hierarchical clustering technique [14].

The identified objects, obtained by the object finder during the top-down analysis method, are shown in Figures 5.13 and 5.14. The groups corresponding to these identified objects are derived after the user performs modifications on the identified objects. Similarly to Section 5.4.1, we name a group based on the name of the object that correspond to the group. The groups are shown in Figures 5.15 and 5.16.

Similarly to Section 5.4.1, the clusters of this test case were identified by a clustering tool called basili [21] based on the hierarchical clustering technique. The identified clusters in the symbol table management program are shown in Figure 5.17.

```
Object z#objTFILE*#3 is {
   ~~FILE*~124                                  :(T)FILE*
   table~___II~output_usage~283                 :(R)output_usage
   table~___II~output_table~271                 :(R)output_table
}
Object z#objTSymbol*#4 is {
   ~~Symbol*~64                                 :(T)Symbol*
   table~___II~where_defined~448                :(R)where_defined
   table~___II~symbol_dup~361                   :(R)symbol_dup
   table~___II~new_symbol~251                   :(R)new_symbol
   table~___II~move_use~244                     :(R)move_use
   table~___II~move_reference~242               :(R)move_reference
   table~___II~move_def~237                     :(R)move_def
   table~___II~merge_type~234                   :(R)merge_type
   table~___II~make_undefined_symbol~211        :(R)make_undefined_symbol
   table~___II~make_symbol~209                  :(R)make_symbol
   table~___II~ismemberof~194                   :(R)ismemberof
   table~___II~isdefined~192                    :(R)isdefined
   table~___II~is_type_alias~190                :(R)is_type_alias
   table~___II~is_macro_name~188                :(R)is_macro_name
   table~___II~insert_use~184                   :(R)insert_use
   table~___II~insert_symbol~182                :(R)insert_symbol
   table~___II~insert_reference~180             :(R)insert_reference
   table~___II~insert_def~178                   :(R)insert_def
   table~___II~get_top_symbol~149               :(R)get_top_symbol
   table~___II~get_entry_string~145             :(R)get_entry_string
   table~___II~get_default~144                  :(R)get_default
   table~___II~find_symbol~132                  :(R)find_symbol
   table~___II~add_type~63                      :(R)add_type
}
Object z#objTToken*#5 is {
   ~~Token*~95                                  :(T)Token*
   table~___II~token_dup~416                    :(R)token_dup
   table~___II~token_cmp~413                    :(R)token_cmp
   table~___II~new_token~252                    :(R)new_token
   table~___II~isthere~197                      :(R)isthere
}
Object z#objTType*#6 is {
   ~~Type*~254                                  :(T)Type*
   table~___II~type_dup~429                     :(R)type_dup
   table~___II~type_cat~426                     :(R)type_cat
   table~___II~new_type~253                     :(R)new_type
}
```

Figure 5.13. Types-based candidate objects identified in symbol table management program

```
Object z#objTchar*#7 is {
   ~~char*~58                            :(T)char*
   table~___II~get_type_name~151         :(R)get_type_name
   table~___II~get_table_index~147       :(R)get_table_index
}
Object z#objTentry_tag*#8 is {
   ~~entry_tag*~250                      :(T)entry_tag*
   table~___II~new_entry~249             :(R)new_entry
}
Object z#objT+undetermined-type#9 is {
   ~~symbo_tag*~208                      :(T)symbo_tag*
   ~~Type*~254                           :(T)Type*
   ~~Token*~95                           :(T)Token*
   ~~Symbol*~64                          :(T)Symbol*
   ~~FILE*~124                           :(T)FILE*
   table~___II~output_use~285            :(R)output_use
   table~___II~output_type~276           :(R)output_type
   table~___II~output_type_member~280    :(R)output_type_member
   table~___II~output_token~273          :(R)output_token
   table~___II~output_reference~268      :(R)output_reference
   table~___II~output_parameter~262      :(R)output_parameter
   table~___II~output_parameter_usage~265  :(R)output_parameter_usage
   table~___II~output_def~259            :(R)output_def
   table~___II~output_declaration_type~256 :(R)output_declaration_type
   table~___II~make_name~205             :(R)make_name
}
```

Figure 5.13 – – continued

```
Object z#objGcurrent_count#2 is {
   table~___II~use_list~446                  :(G)use_list
   table~___II~symbol_table~363              :(G)symbol_table
   table~___II~goto_reference_list~157       :(G)goto_reference_list
   table~___II~def~98                        :(G)def
   table~___II~debug_token~94                :(G)debug_token
   table~___II~current_side~92               :(G)current_side
   table~___II~current_scope~91              :(G)current_scope
   table~___II~current_func_name~90          :(G)current_func_name
   table~___II~current_count~89              :(G)current_count
   table~___II~where_defined~448             :(R)where_defined
   table~___II~table_reset~382               :(R)table_reset
   table~___II~table_init~381                :(R)table_init
   table~___II~table_final~380               :(R)table_final
   table~___II~remove_scope_flag~310         :(R)remove_scope_flag
   table~___II~pr_debug_token~294            :(R)pr_debug_token
   table~___II~output_usage~283              :(R)output_usage
   table~___II~output_table~271              :(R)output_table
   table~___II~make_symbol~209               :(R)make_symbol
   table~___II~make_name~205                 :(R)make_name
   table~___II~isdefined~192                 :(R)isdefined
   table~___II~is_type_alias~190             :(R)is_type_alias
   table~___II~is_macro_name~188             :(R)is_macro_name
   table~___II~insert_use~184                :(R)insert_use
   table~___II~insert_symbol~182             :(R)insert_symbol
   table~___II~insert_reference~180          :(R)insert_reference
   table~___II~insert_def~178                :(R)insert_def
   table~___II~flag_scope_flag~136           :(R)flag_scope_flag
   table~___II~find_symbol~132               :(R)find_symbol
}
```

Figure 5.14. Globals-based candidate objects identified in symbol table management program

```
Group z#objTFILE*#3 is {
   ~~FILE*~124                              :(T)FILE*
   table~___II~output_usage~283             :(R)output_usage
   table~___II~output_table~271             :(R)output_table
}
Group z#objTSymbol*#4 is {
   ~~Symbol*~64                             :(T)Symbol*
   table~___II~symbol_dup~361               :(R)symbol_dup
   table~___II~new_symbol~251               :(R)new_symbol
   table~___II~move_use~244                 :(R)move_use
   table~___II~move_reference~242           :(R)move_reference
   table~___II~move_def~237                 :(R)move_def
   table~___II~merge_type~234               :(R)merge_type
   table~___II~make_undefined_symbol~211    :(R)make_undefined_symbol
   table~___II~make_symbol~209              :(R)make_symbol
   table~___II~ismemberof~194               :(R)ismemberof
   table~___II~isdefined~192                :(R)isdefined
   table~___II~is_type_alias~190            :(R)is_type_alias
   table~___II~is_macro_name~188            :(R)is_macro_name
   table~___II~insert_use~184               :(R)insert_use
   table~___II~insert_symbol~182            :(R)insert_symbol
   table~___II~insert_reference~180         :(R)insert_reference
   table~___II~insert_def~178               :(R)insert_def
   table~___II~get_top_symbol~149           :(R)get_top_symbol
   table~___II~get_entry_string~145         :(R)get_entry_string
   table~___II~get_default~144              :(R)get_default
   table~___II~find_symbol~132              :(R)find_symbol
   table~___II~add_type~63                  :(R)add_type
}
Group z#objTToken*#5 is {
   ~~Token*~95                              :(T)Token*
   table~___II~token_dup~416                :(R)token_dup
   table~___II~token_cmp~413                :(R)token_cmp
   table~___II~new_token~252                :(R)new_token
   table~___II~isthere~197                  :(R)isthere
}
Group z#objTType*#6 is {
   ~~Type*~254                              :(T)Type*
   table~___II~type_dup~429                 :(R)type_dup
   table~___II~type_cat~426                 :(R)type_cat
   table~___II~new_type~253                 :(R)new_type
}
```

Figure 5.15. Groups based on types-based objects identified in symbol table management program

```
Group z#objTchar*#7 is {
   ~~char*~58                                   :(T)char*
   table~___II~get_type_name~151                :(R)get_type_name
   table~___II~get_table_index~147              :(R)get_table_index
}
Group z#objTentry_tag*#8 is {
   ~~entry_tag*~250                             :(T)entry_tag*
   table~___II~new_entry~249                    :(R)new_entry
}
Group z#objT+undetermined-type#9 is {
   ~~symbo_tag*~208                             :(T)symbo_tag*
   ~~Type*~254                                  :(T)Type*
   ~~Token*~95                                  :(T)Token*
   ~~Symbol*~64                                 :(T)Symbol*
   ~~FILE*~124                                  :(T)FILE*
   table~___II~output_use~285                   :(R)output_use
   table~___II~output_type~276                  :(R)output_type
   table~___II~output_type_member~280           :(R)output_type_member
   table~___II~output_token~273                 :(R)output_token
   table~___II~output_reference~268             :(R)output_reference
   table~___II~output_parameter~262             :(R)output_parameter
   table~___II~output_parameter_usage~265       :(R)output_parameter_usage
   table~___II~output_def~259                   :(R)output_def
   table~___II~output_declaration_type~256 :(R)output_declaration_type
}
```

Figure 5.15 – – continued

```
Group z#objGcurrent_count#2 is {
   table~___II~use_list~446                :(G)use_list
   table~___II~symbol_table~363            :(G)symbol_table
   table~___II~goto_reference_list~157     :(G)goto_reference_list
   table~___II~def~98                      :(G)def
   table~___II~debug_token~94              :(G)debug_token
   table~___II~current_side~92             :(G)current_side
   table~___II~current_scope~91            :(G)current_scope
   table~___II~current_func_name~90        :(G)current_func_name
   table~___II~current_count~89            :(G)current_count
   table~___II~where_defined~448           :(R)where_defined
   table~___II~table_reset~382             :(R)table_reset
   table~___II~table_init~381              :(R)table_init
   table~___II~table_final~380             :(R)table_final
   table~___II~remove_scope_flag~310       :(R)remove_scope_flag
   table~___II~pr_debug_token~294          :(R)pr_debug_token
   table~___II~make_name~205               :(R)make_name
   table~___II~flag_scope_flag~136         :(R)flag_scope_flag
}
```

Figure 5.16. Groups based on globals-based objects identified in symbol table management program

Figure 5.17 showed the clusters defined on this example using Basili's clustering techniques. The groups corresponding to these clusters are shown in Figure 5.18. In this case, a group consists of one or more clusters of routines that maintain the same levels of coupling and strength [14] of the cluster identified with hierarchical clustering technique. The groups are similarly constructed to the groups for the clusters identified in the algebraic expression evaluation program of Section 5.4.2. Initially, groups are defined based on the top level of the corresponding clusters. Then, the routines which did not cluster were grouped into a single group, named group `V`.

5.5 Comparison of Complexity

This section presents the results of the comparison of the three test cases in study I and a summary of the results for all three test cases. The test cases are the

```
Final dendrogram:
 -- Cluster No.1 --
(3 merge_type
   type_cat
   output_parameter
   output_table
   output_reference
   output_type_member)

 -- Cluster No.2 --
(14 get_top_symbol
    symbol_dup
    token_dup
    type_dup
    (8 insert_def
       move_def)
    table_final
    (8 insert_use
       move_use)
    (6 isthere
       move_reference)
    output_def
    output_parameter_usage
    output_usage
    output_use)
```

```
 -- Cluster No.3 --
(100 (50 find_symbol
         insert_symbol
         get_entry_string
         get_table_index)
     (38 insert_reference
         ismemberof)
     get_type_name
     isdefined
     make_undefined_symbol
     where_defined)
```

Figure 5.17. Clusters found in symbol table management program

```
Group I is {
   table~___II~merge_type~234                 :(R)merge_type
   table~___II~type_cat~426                   :(R)type_cat
   table~___II~output_parameter~262           :(R)output_parameter
   table~___II~output_table~271               :(R)output_table
   table~___II~output_reference~268           :(R)output_reference
   table~___II~output_type_member~280         :(R)output_type_member
}
Group II is {
   table~___II~get_top_symbol~149             :(R)get_top_symbol
   table~___II~symbol_dup~361                 :(R)symbol_dup
   table~___II~token_dup~416                  :(R)token_dup
   table~___II~type_dup~429                   :(R)type_dup
   table~___II~insert_def~178                 :(R)insert_def
   table~___II~move_def~237                   :(R)move_def
   table~___II~table_final~380                :(R)table_final
   table~___II~insert_use~184                 :(R)insert_use
   table~___II~move_use~244                   :(R)move_use
   table~___II~isthere~197                    :(R)isthere
   table~___II~move_reference~242             :(R)move_reference
   table~___II~output_use~285                 :(R)output_use
   table~___II~output_parameter_usage~265     :(R)output_parameter_usage
   table~___II~output_usage~283               :(R)output_usage
   table~___II~output_def~259                 :(R)output_def
}
Group III is {
   table~___II~find_symbol~132                :(R)find_symbol
   table~___II~insert_symbol~182              :(R)insert_symbol
   table~___II~get_entry_string~145           :(R)get_entry_string
   table~___II~get_table_index~147            :(R)get_table_index
   table~___II~insert_reference~180           :(R)insert_reference
   table~___II~ismemberof~194                 :(R)ismemberof
   table~___II~get_type_name~151              :(R)get_type_name
   table~___II~isdefined~192                  :(R)isdefined
   table~___II~make_undefined_symbol~211      :(R)make_undefined_symbol
   table~___II~where_defined~448              :(R)where_defined
}
```

Figure 5.18. Groups based on clusters found in symbol table management program

```
Group IV is {
   table~___II~new_symbol~251                  :(R)new_symbol
   table~___II~make_symbol~209                 :(R)make_symbol
   table~___II~is_type_alias~190               :(R)is_type_alias
   table~___II~is_macro_name~188               :(R)is_macro_name
   table~___II~get_default~144                 :(R)get_default
   table~___II~add_type~63                     :(R)add_type
   table~___II~token_cmp~413                   :(R)token_cmp
   table~___II~new_token~252                   :(R)new_token
   table~___II~new_type~253                    :(R)new_type
   table~___II~new_entry~249                   :(R)new_entry
   table~___II~output_type~276                 :(R)output_type
   table~___II~output_token~273                :(R)output_token
   table~___II~output_declaration_type~256     :(R)output_declaration_type
   table~___II~table_reset~382                 :(R)table_reset
   table~___II~table_init~381                  :(R)table_init
   table~___II~remove_scope_flag~310           :(R)remove_scope_flag
   table~___II~pr_debug_token~294              :(R)pr_debug_token
   table~___II~make_name~205                   :(R)make_name
   table~___II~flag_scope_flag~136             :(R)flag_scope_flag
}
```

Figure 5.18 – – continued

name cross-reference program of Section 5.4.1, the algebraic expression evaluation of Section 5.4.2, and the symbol table management program of Section 5.4.3.

The comparison methodology consisted of comparing the complexity of two partitionings of each test case, as expressed in terms of the primitive complexity metrics factors. Then, the summary of all three test cases is presented factor by factor at the end of this section.

First, the primitive factors were computed for each partitioning of a test case program. A partitioning is due to a particular modularization approach; for this study, we considered two approaches to obtain the partitioning: our object finder and the hierarchical clustering technique by Hutchens and Basili [14]. Hierarchical clustering was chosen for the study since this modularization technique is also based on data bindings found in a program; their approach is one of several modularization (clustering) techniques which were explained in Section 2.1.1.1. Each partitioning consists of a set of groups derived in Section 5.4. Second, the comparison of the complexity of the two partitionings of a test case consists of comparing one complexity metrics factor at a time.

The results are presented for each test case individually. We computed the primitive complexity factors for the groups abstracted from the identified objects and also for the groups abstracted from the clusters defined using Basili's clustering technique. Then, we illustrate a factor by factor interpretation of the observed results for each test case. At the end of this section, we present an overall interpretation of the study which consists of summarizing the results of the three test cases. This interpretation of the summary results demonstrates that the object finder approach results in a modularization of a program which exhibits significantly lower inter-group complexity and moderately lower intra-group complexity compared to hierarchical clustering techniques.

5.5.1 Test Case 1: Name Cross-reference Program

This section presents the comparison of the metrics factors computed for both versions of the name cross-reference program. Version 1 is the partitioning (groups) derived using the objects finder and version 2 is the partitioning (groups) derived using hierarchical clustering techniques.

First, the comparison of inter-group complexity metrics.

- Factors f_1 and f_2, the sets of direct interface variables and indirect interface variables, respectively, are not used in this comparison since there are few (only one) global variables to consider in the program.

- Factor f_3, the boundary set, shows the similar inter-group complexity observed among the groups derived from both clustering techniques since the average number of data bindings in version 1 was 3 whereas the average number of data bindings in version 2 was 3.5. On the other hand, the inter-group complexity of the interface can be enhanced by the object finder as shown by the reduced number of global variables in the boundary set shown by factor f_4 with a total of 10 variables in version 1 and, 12 variables in version 2.

- Factors f_5 and f_6, the sum of type sizes of direct and indirect interface variables, were not used in this comparison since there are few (only one) global variables to consider in this program.

- Factors f_7 and f_8, the sum of the type sizes of boundary set variables and the set of types of boundary set variables, respectively, showed that on the average groups derived from Basili's clustering techniques tend to exhibit a more complex interface since their routines "manipulate" a larger set of types. From factor f_7, we determined the total type size of program version 1 and

2 to be 30 and 98, respectively. Also, from factor f_8, the number of types manipulated by the interface of groups in version 2 was larger than the number of types manipulated by the interface of groups in version 1. Thus, the inter-group complexity was larger between the groups of version 2.

Second the intra-group complexity factors are analyzed.

- Factors f_9 and f_{10}, the set of direct internal variables and indirect internal variables, respectively, were not used in this comparison since there were few (only one) global variables to consider in the program.

- Factor f_{11}, the interior set, showed similar intra-group complexity between the groups in both versions of the program as observed from the number of bindings in versions 1 and 2. Also, factor f_{12}, the percentage of modules accessing all global variables, illustrated the similarity of intra-group complexity between the two program versions.

- Factors f_{13} and f_{14}, the sum of type sizes of interior set variables and the set of types of interior set variables, respectively, reflected the larger intra-group complexity observed in the groups derived from Basili's clustering techniques; specifically, from factor f_{13}, we determined that the total type size of program version 1 and 2 were 26 and 50, respectively. Also, from factor f_{14}, the number of types manipulated by routines in groups of version 1 was equal to the number of types manipulated by routines in groups of version 2. Even though a similar set of types was manipulated by groups in each version, the group's intra-group complexity was different due to the frequency of use of those types.

Finally, the intra-group strength was not considered in the comparison since Basili's clustering technique does not use type information during the clustering.

This test case shows some of the advantages of using the object finder to modularize a system. The resulting partitioning exhibited significantly reduced inter-group complexity and reduced intra-group complexity when compared to the partitioning resulting from hierarchical clustering.

5.5.2 Test Case 2: Algebraic Expression Evaluation Program

This section presents the comparison of the metrics factors computed for both versions of the algebraic expression evaluation program. Version 1 is the partitioning (groups) derived using the objects finder and version 2 is the partitioning (groups) derived using hierarchical clustering techniques.

First, the comparison of inter-group complexity metrics.

- Factors f_1 and f_2, the sets of direct interface variables and indirect interface variables, respectively, show the reduced inter-group complexity of the identified objects in terms of the number of interface variables since version 1 had no interface variable and version 2 had an average of three interface variables between groups.

- Factor f_3, the boundary set, illustrates the reduced inter-group complexity between the groups of version 1 compared to the groups of version 2 in terms of the number of data bindings between groups since the average number of data bindings between the groups in version 1 was 9.8 data bindings whereas between groups in version 2 was 23.4 data bindings. That is also the case for factor f_4, the set of variables in the boundary set. The average number of variables in the boundary set for version 1 was 7.8 variables whereas the average for version 2 was 15.

- Factors f_5 and f_6, the sum of type sizes of direct and indirect interface variables, showed the reduced the inter-group complexity of the identified objects in terms of the type size of the interface variables since version 1 type size was zero and version 2 had a maximum type size of 106 between its groups. This showed the reduced complexity due to the interface variables.

- Factor f_7, the sum of type sizes of boundary set variables, showed the reduced inter-group complexity of the identified objects in terms of the total type size of the variables defining data bindings between groups since the total type size of version 1 was 63 whereas the total type size of version 2 was 277.

- Factor f_8, the set of types of boundary set variables, showed the reduced inter-group complexity of the identified objects in terms of the number of different types manipulated by the interface of groups. On the average, the number of types manipulated by the interface of groups in version 1 was slightly less than the number of types manipulated by the interface of groups in version 2.

Second the intra-group complexity factors are analyzed.

- Factors f_9 and f_{10}, the set of direct internal variables and indirect internal variables, respectively, showed the increased intra-group complexity in the identified objects in terms of the number of internal variables since version 1 had more internal variables inside the groups than version 2. However, this is a signal of increased strength of the groups in version 1.

- Factor f_{11}, the interior set, showed the slightly increased intra-group complexity of the identified objects in terms of the number of data bindings inside the groups since the maximum number of data bindings inside the groups in version 1 was 55 whereas the maximum number of data bindings inside groups

in version 2 was 30. Thus, the intra-group complexity was slightly larger in version 1 than in version 2.

- Factor f_{12}, the percentage of modules accessing all global variables, was not used in this comparison since it can only be computed for those groups derived from globals-based candidate objects.

- Factor f_{13}, the sum of type sizes of interior set variables, showed slightly increased intra-group complexity of the identified objects in terms of the total type size of program version 1 and 2 since the total type size of version 1 was 326 whereas the total type size of version 2 was 226.

- Factor f_{14}, the set of types of interior set variables, showed similar intra-group complexity inside the groups in both versions of the program as observed from the number of different types manipulated by routines in groups of version 1 and the number of different types manipulated by routines in groups of version 2.

Finally, the intra-group strength factors were not considered in the comparison since Basili's clustering technique does not use type information during the clustering.

5.5.3 Test Case 3: Symbol Table Management Program for Acx

This section presents the comparison of the metrics factors computed for both versions of the symbol table management program. Version 1 is the partitioning (groups) derived using the objects finder and version 2 is the partitioning (groups) derived using hierarchical clustering techniques.

First, the comparison of inter-group complexity metrics.

- Factors f_1 and f_2, the sets of direct interface variables and indirect interface variables, respectively, show the increased inter-group complexity of the identified objects since the number of interface variables in version 1, equal to 3, was larger than the number of interface variables in version 2, equal to 1.

- Factor f_3, the boundary set, shows the significantly reduced inter-group complexity between the groups in version 1 compared to the inter-group complexity of groups in version 2 in terms of the number of data bindings between groups since the average number of data bindings between the groups in version 1 is 26.6 data bindings whereas between the groups in version 2 is 41 data bindings. However, the maximum number of data bindings was much larger (80) in version 1 than in version 2 (51). The inter-group complexity was also reduced, as illustrated by factor f_4, the set of variables in the boundary set. The average number of variables in the boundary set for version 1 was 13.2 whereas version 2's average was 26.7.

- Factors f_5 and f_6, the sum of type sizes of direct and indirect interface variables, showed the slightly increased inter-group complexity of the identified objects in terms of the type size of interface variables since the average sum of type sizes of interface variables between groups in version 1 (56.3) was slightly higher than the average sum of type sizes of interface variables between groups in version 2 (55).

- Factor f_7, the sum of type sizes of boundary set variables, showed the reduced inter-group complexity of the identified objects in terms of the total type size between the groups since the total type size of version 1 was 267 and the total type size of version 2 was 323.

- Factor f_8, the set of types of boundary set variables, showed the reduced inter-group complexity of the identified objects in terms of the number of different types manipulated by the interface of groups. On the average, the number of types manipulated by the interface of groups in version 1 (3.25) was less than the number of types manipulated by the interface of groups in version 2 (6.2).

Second the intra-group complexity factors are analyzed.

- Factors f_9 and f_{10}, the set of direct internal variables and indirect internal variables, respectively, showed similar intra-group complexity inside the groups in both versions of the program as observed from the number of internal variables in version 1 and 2.

- Factor f_{11}, the interior set, showed reduced intra-group complexity of the identified objects in terms of the number of data bindings inside groups since the average number of data bindings in the groups of version 1 (10.7) was more than half the average number of data bindings in the groups of version 2 (26.5).

- Factor f_{12}, the percentage of modules accessing all global variables, was not used in this comparison since it can only be computed for those groups derived from globals-based candidate objects.

- Factor f_{13}, the sum of type sizes of interior set variables, showed similar intra-group complexity inside the groups since the total type size of program version 1 and 2 was very close; specifically, the total type size of version 1 was 157 and the total type size of version 2 was 167.

- Factor f_{14}, the set of types of interior set variables, showed similar intra-group complexity inside the groups in both versions of the program as observed from

the number of different types manipulated by routines in groups of version 1 and the number of different types manipulated by routines in groups of version 2.

Finally, the intra-group strength factors were not considered in the comparison since Basili's clustering technique does not use type information during the clustering.

5.5.4 Summary and Conclusions of the Comparison

This section presents the summary of the three test cases presented in the previous sections for each of the primitive factors of complexity which affects inter-group and intra-group complexity. We assume that version 1 consists of the partitioning (groups) resulting from the object finder, i.e., the identified objects, and version 2 corresponds to the partitioning (groups) resulting from hierarchical clustering, i.e.: the clusters.

The factors for inter-group complexity are:

- Factors f_1 and f_2, the sets of direct interface variables and indirect interface variables, respectively, included a test case in which the inter-group complexity was higher in version 1 than in version 2, and another test case in which the inter-group complexity was higher in version 2 than in version 1; this factor was ignored in the third test case. Since the differences in the values of these factors were not significant, we concluded that both versions had similar inter-group complexity based on factors f_1 and f_2 in all three test cases.

- Factor f_3, the boundary set, showed significantly reduced inter-group complexity between the groups in one test cases and, at least, somewhat reduced inter-group complexity in another test cases. This factor demonstrated that the object finder-based groups presented lower inter-group complexity than hierarchical clustering-based groups.

- Factor f_4, the set of variables in the boundary set, showed that the object finder significantly reduced the inter-group complexity between the groups in all three test cases. In two test cases, the average number of variables in the boundary set was reduced by half.

- Factors f_5 and f_6, the sum of type sizes of direct and indirect interface variables, showed significantly reduced inter-group complexity in the identified objects in one of the test cases. However, in another test case, the inter-group complexity slightly increased in the groups derived from identified objects. This factor showed the reduced inter-group complexity of the identified objects.

- Factor f_7, the sum of type sizes of boundary set variables, showed significantly reduced inter-group complexity of the identified objects in terms of the total type size of the variables in the boundary set between groups in all three test cases.

- Factor f_8, the set of types of boundary set variables, showed significantly reduced inter-group complexity of the identified objects in terms of the number of different types manipulated by the interface of groups in all three test cases.

The factors for intra-group complexity are:

- Factors f_9 and f_{10}, the set of direct internal variables and indirect internal variables, respectively, included a test case in which the intra-group complexity was higher in version 1 than in version 2, and another test case in which the intra-group complexity was similar in both versions; these factors were ignored in the third test case. The three test cases did not illustrate a conclusive behavior of the intra-group complexity expressed by these factors.

- Factor f_{11}, the interior set, included a test case in which the intra-group complexity was higher in version 1 than in version 2 and another test case in which the intra-group complexity was higher in version 2 than in version 1; in the third test case, this factor was similar in both versions. We concluded that on the average, this factor was similar in both versions and, in fact, we should observe a small reduction on intra-group complexity measured by this factor when an statistically significant population is used for the comparison.

- Factor f_{12}, the percentage of modules accessing all global variables, was not used in the comparison since it can only be computed for those groups derived from identified objects using the globals-based analysis algorithm.

- Factor f_{13}, the sum of type sizes of interior set variables, included a test case in which the intra-group complexity was higher in version 1 than in version 2, and another test case in which the intra-group complexity was higher in version 2 than in version 1; the third test case showed slightly reduced intra-group complexity in the identified objects. We concluded that factor f_{13} showed slightly reduced intra-group complexity in the groups derived from identified objects.

- Factor f_{14}, the set of types of interior set variables, showed similar intra-group complexity inside the groups in both versions of the program as observed from the number of different types manipulated by routines in each version in all three test cases.

The data from the three test cases in study I was summarized above, and shows some evidence of the effectiveness of the object finder to find groups that exhibit desired inter-group and intra-group complexity levels as compared to hierarchical

clustering techniques. It also shows the trend of the results expected from the future evaluation of the approach.

The data illustrates that three out of eight inter-group complexity factors showed significantly reduced complexity, three other inter-group complexity factors showed reduced inter-group complexity, and the two remaining inter-group complexity factors showed similar inter-group complexity. Furthermore, the data illustrates that two out of six intra-group complexity factors showed slightly reduced intra-group complexity, one intra-group complexity factor showed similar intra-group complexity, and the three other intra-group complexity factors were not used in the study.

The implications of this study are that the object finder approach results in a modularization of a program which exhibited significantly lower inter-group complexity and moderately lower intra-group complexity when compared to hierarchical clustering techniques. It is important to note that those complexity metrics factors that measure the intra-group strength were not considered during this study since they measure characteristics of a program which are not considered by the hierarchical clustering approach [14]. Hence, this reduced the possibility that the complexity metrics factors might be used to favor the object finder approach.

In this interpretation of the data in Study I, the primitive complexity metrics factors were equally weighted with respect to the total complexity of a group. During the future study, the factors will not be evenly weighted when computing the total complexity. In fact, one of the proposed future research consists of expressing a mathematical relation between these complexity metrics factors. In Section 5.3, we explained the intuitive effect of each factor on the total complexity. The mathematical relationship expressing the total complexity will be obtained by specifying the weights of each complexity metrics factors on the total complexity.

CHAPTER 6
APPLICATIONS: DESIGN RECOVERY BASED ON THE APPROACH

This chapter illustrates the potential usefulness of our approach by applying it in the context of design recovery.

As shown in Figure 6.1, there are two major approaches of system modularization in the object finder. The input in each case is a "C" program. The *top-down* approach decomposes a whole system into a set of related components. The output of the top-down approach is a collection of candidate objects. The *bottom-up* approach starts from a user-identified subcomponent and grows up until all related subcomponents are also collected. The output of the bottom-up approach is an extended candidate object. The top-down approach provides an overall understanding of the entire system. On the other hand, the bottom-up approach provides a view of the information which is closely related to the components which are under examination.

The process of object identification can be performed top-down which results in a modularization of a system into recursively containing objects (the objects are themselves modularized into other set of objects). The kind of design information obtained by this process corresponds to the structure of the program in terms of its components and the relationships between those components. Next, we present the methods that summarize these approaches of system modularization.

<u>Method 1</u> *Top-down Analysis*

Input : A candidate object; for the complete system: the system object, otherwise a simple object.

Output : A collection of candidate objects.

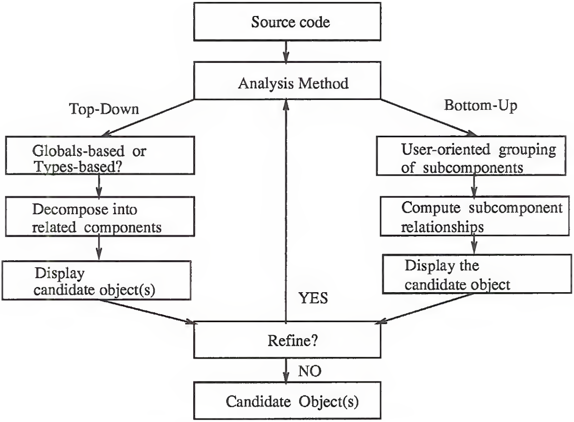

Figure 6.1. The object finder system flow

Steps :

1. *Perform either or both of Globals-based Object Finder (Algorithm 1) or Types-based Object Finder (Algorithms 2).*
2. *Decompose the current candidate object into identified candidate objects.*
3. *Display to the user the identified candidate objects.*
4. *Refine a selected candidate object and store the identified candidate objects for future reference.*

This approach is complemented with heuristics and human input. These heuristics will be abstracted from the human input during the future study and will eventually replace the human input in the object finder. The kind of heuristics in this analysis are similar to those utilized by a designer when a program decomposition is first defined.

Analysis. The time complexity associated with this method depends on the complexity of the Globals-based and Types-based Object Finder algorithms. Each level of modularization will decrease the size of the identified objects in terms of the number of routines, global variables, and (abstract data) types; then, the required time to analyze an object will also decrease. The display time of the identified objects is proportional to the number of identified objects at each level of decomposition.

The process of object identification could also be as follows:

Method 2 Bottom-up Analysis

Input : An initial candidate object defined by the user

Output : The extended candidate object.

Steps :

1. *Let the user propose a starting point—one or more routines, types, or data items to be part of an initial candidate object* (F, T, D).
2. *Let the user choose if we will next apply* data–routine *analysis,* routine–data *analysis,* type–routine *analysis, or* routine–type *analysis:*
   (a) *If he chooses* data–routine *analysis: add to* F *the set of all routines that use data items in* D.
   (b) *If he chooses* routine–data *analysis: add to* D *the set of all global and persistent data items used by routines in* F.
   (c) *If he chooses* type–routine *analysis: add to* F *the set of all routines that use types in* T *for formal parameters or return values.*
   (d) *If he chooses* routine–type *analysis: add to* T *the set of all types used as formal parameters or return values by the routines in* F.
3. *Draw out a graph showing the* F, T *and* D *and their connections based on globals-based analysis (Algorithm 1) and types-based analysis (Algorithm 2) for the user to review. He may delete any of the components he wishes and either repeat the previous step or store the candidate object for future reference.*

Analysis. This method is an iterative approach to group the related components in a system into a single candidate object. The initial candidate object is determined by the user. Let the number of routines proposed by the user at the starting point be r, the number of types be t, and the number of global variables be

v. The complexity of the initial step depends on the kind of analysis chosen by the user as follows:

If the user chooses	worst-case time complexity
Data-routine analysis	$O(N.v)$
Routine-data analysis	$O(r.g)$
Type-routine analysis	$O(R.t)$
Routine-type analysis	$O(r.T)$

where N is the number of routines in the system, g is the number of global variables in the system, and T is the number of (abstract data) types in the system.

The next step consists of calculating the connections between components of a system using the Globals-based and Types-based Object Finder algorithms. In Chapter 4, the bounding time complexities of these two algorithms were $O(g^2N^2)$ and $O(max(T^2, NT))$, respectively.

Chapter 8 reports on the usefulness of these methods for recovering the design information. Section 8.1 illustrates the Top-down Analysis method and Section 8.3 illustrates the Bottom-up Analysis method.

CHAPTER 7
A PROTOTYPE FOR THE PROPOSED APPROACH

This chapter outlines an interactive prototype tool which implements the approach. This tool was designed to demonstrate the flexibility and portability of its design.

The object finder implements both Top-down and Bottom-up Analysis (Methods 1 and 2) as well as the Globals-based and Types-based Object Finder (Algorithms 1 and 2) in the context of a design recovery tool. Appendix A presents the user's manual of the object finder prototype.

Top-down Analysis could well be the most productive way to get a first cut at the object content of an entire software system, especially if heuristics can be developed to speed or replace human intervention. However, both algorithms could also be used "bottom up" and together; this may be the most useful way to organize an exploratory object finding tool.

7.1 The Object Finder: A GNU Emacs-based Design Recovery Tool

A conceptual model of the object finder is presented in Figure 7.1 which shows the interactions between the object finder and the environment, including a user, a graphics interface, a program base, an object base, and Emacs. The communication consists of the command control flow, issued by the user, and data flows for access to the program and object bases. The program to be analyzed is read in from the program base. Currently, the prototype analyzes a "C" program that consists of a collection of source files and a make file which describes how the program is constructed. The user may choose to use Emacs for any program modification, to use the object finder to identify high level objects, or to use the graphics interface

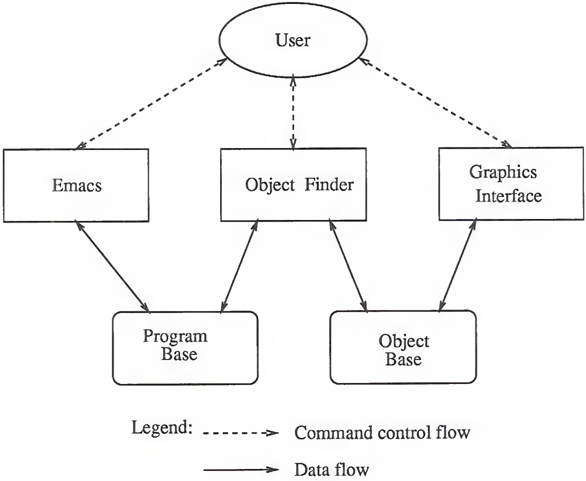

Figure 7.1. The object finder conceptual model

to display the objects on a Sun workstation. The object base consists of the set of candidate objects identified by the object finder.

7.2 Design Goals of the Object Finder

The object finder was implemented to achieve several design goals including ease of use, portability, and flexibility. The GNU Emacs programming environment provided a prototyping system with these and other attributes [40]. GNU Emacs is an extensible editor which derives much of its power and flexibility from its organization. As illustrated in Figure 7.2, we use a similar organization in the object finder. In the figure, the object finder implementation consists of the functions implemented in GNU Emacs LISP, the keyboard interpreter which provides functionality for capturing user's input from the keyboard, and the display routines which present output to the user's terminal. In addition, Emacs interacts with external processes such as a cross-reference generator and a graphics user interface.

The internal structure of Emacs was used to organize the functions of the object finder. Processes were used to control the activities of the object finder, buffers were used as data structures to store the information it generated, and windows were used to display the text output on the user's terminal. The steps of the object finder are as follows: First, the object finder is initialized by creating the symbol table of the program. Second, the "C" program is decomposed into identified objects using the proposed algorithms, and the relationships among objects are derived. Third, text output is generated, stored into the output buffers, and displayed in the windows. Finally, the identified objects and the relationships are graphically displayed on an X-window user interface [28].

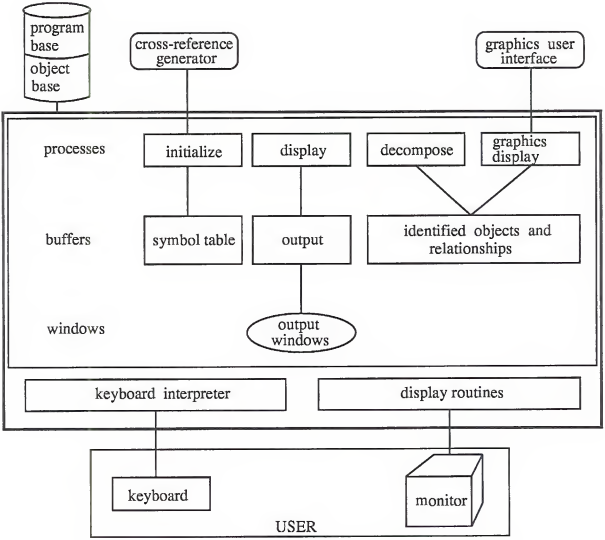

Figure 7.2. The object finder implementation outline

7.3 Design of the Object Finder

The design of the object finder was developed in several stages. The initial design was the kernel of the object finder containing basic IO functions which were separated from other components to minimize the risks of personnel changeovers.

An example of these basic IO functions is the following set of Emacs LISP routines in Figure 7.3 which implement a scanner of tokens in a given input stream. The program under analysis consists of several kind of identifiers including variables, types, routines, reserved words, and file names, as well as numeric and string constants. In this example, function `get-token` gets the next available token from the input which is a "C" program. If a token has been previously recognized from the input, the global variable `token_p` is true and the current value of the global variable `token` is returned; otherwise, a new token is scanned from the input stream. The function `succ-token` looks ahead to the next input token. The function `get-token0` scans the input stream for available tokens; this function recognizes the identifiers and operators in a C input program. The first step in the functions is to determine the input stream (a buffer) by using the function `set-buffer` to set the current buffer to the buffer named FROM. The function `more` forms a token from the input characters. The function `member` determines set membership of its first parameter in its second parameter.

Several other functions were provided to support the previous functions including functions to read a character from the input (`get-byte`); to determine whether a character is an alphabetic character (`alpha-c`), a digit character (`digit-c`), an end-of-buffer mark (`end-of-buffer`), or a blank (including tabs, spaces, and end-of-lines) (`blank`); to look ahead at the next input character (`succ-char`); and to retrieve the next character as a number (`get-char`). Finally, the function `newlines`

adds a newline mark to each file in the list of files FILES, the function `skip-line` moves the start of next line in the buffers, and the function `add-lineno2` increments the line counter in the buffer FROM.

One of the most interesting design decisions was the use of Emacs buffers for implementing most of the interfaces among the components of the object finder. A buffer holds edited data in Emacs and could be associated with one or more windows. Buffers are used as an interface to store the program under analysis, the set of identified candidate objects, the data generated during the analysis, and the information to be displayed. In Figure 7.1, the interface between the program base and the object finder was implemented in terms of buffers as well as the interfaces between the object base and the object finder, and between the object base and the graphics user interface.

Another design decision was the use of processes to handle several external functions. Processes are primarily used to interact with external programs that generate the program symbol table and the cross reference information, or to provide the graphics interface.

For example, a process is used to link to the ANSI "C" cross-reference tool. Two functions are used to control this process; Figure 7.4 shows part of the process-control code in one of the functions. A process communicates with the object finder through buffers. Function `start-process` starts a program and returns the process identifier [20]. The input buffer, `program-file`, to this process consists of the "C" source code for which cross-reference information is needed and the output buffer, `cross-ref-set`, will contain this information. The external program invoked by this process is called `acx`. Finally, the `data-file-name-directory` specifies the data directory where the source files are located.

```
(defun get-token (from &optional lf files)
  "Get next token from buffer FROM. LF is non nil if the newline mark
   is treated as a token. FILES is a list of files associated with
   the input stream."
  (set-buffer from)
  (let ((s (get-token0 from lf files)))
    (cond ((equal s ''/*'') (more 'comment from)
                            (get-token from lf files))
          (t s))))

(defun get-token0 (from lf files)
  "Get next token from buffer FROM; move point too. LF=t if a newline
   mark is considered as a token."
  (set-buffer from)
  (cond (token_p (setq token_p nil) token)
   ((end-of-buffer from) nil)
   ((and lf (= 10 (succ-char from))) (get-byte from)
      (add-lineno2 from) (newlines files) (get-token0 from lf files))
   ((blank (succ-char from)) (get-char from)
      (get-token0 from lf files))
   ((= (succ-char from) 10) (get-char from) (add-lineno2 from)
      (get-token0 from lf files))
   ((alpha (succ-char from)) (more 'alphas from))
   ((digit (succ-char from)) (more 'digits from))
   ((member (succ-char from) '(?= ?* ?% ?^ ?!)) (more '= from))
   ((member (succ-char from) '(?-)) (more '- from)) ;"-|--|-=|->"
   ; For simplicity, only operators "-", "--", "-=", and "->" are
   ; included. Other operators such as "/" "/=" "/*" "+" "++"
   ; "+=" "&" "&&" "&=" "|" "||" "|=" "<" "<=" "<<" "<<=" ">" ">="
   ; ">>" ">>=" "." "...", can be handled in a similar fashion.
   (t (get-byte from))))

(defun succ-token (from &optional lf files)
  "Look ahead the next token in buffer FROM."
  (set-buffer from)
  (cond (token_p token)
    (t (setq token (get-token from lf files)) (setq token_p t)
                                               token)))
```

Figure 7.3. A scanner of tokens

```
(start-process "process-name" cross-ref-set "acx"
  "-c" (concat data-file-name-directory program-file))
```

Figure 7.4. Process to control the ANSI "C" cross-reference tool acx

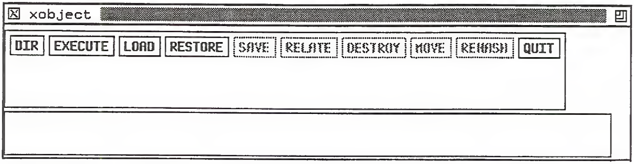

Figure 7.5. `Xobject` commands

Emacs buffers hold processed data, including the symbol table, the identified candidate objects, the graphs and matrices used during the analysis of a program, and the display information. The symbol table is implemented as a set of unique ID's, one for each identifier in the program. The property lists of GNU Emacs LISP were used to describe the characteristics of identifiers.

User interfaces have been recognized as one the most critical parts of a system. The user interface of the object finder consists of menu-driven as well as graphics components. The input of the menu-driven interface was implemented by using an Emacs minibuffer. Output information is displayed in a pre-set window layout which may be adjusted by the user. The graphics user interface provides methods to display and modify the identified objects and their relationships. This interface is explained in the following section.

7.4 `Xobject`: Graphical User Interface

`Xobject` is invoked by the user in the object finder. It allows a user to render a graphics representation of the identified objects, modify the objects, and save the modified objects for later use by the object finder. The first step in `xobject` is to load the candidate objects. The top part of `xobject` display consists of the menu

which is listed in Figure 7.5. `DIR` allows you to set the working directory. `LOAD` loads a file to be analyzed. After a `LOAD`, the identified objects, along with their relationships, are shown. `MOVE` allows you to modify the arrangement of the layout of the objects.

Other `xobject` commands include to temporarily hide an object and all its relationships, to permanently remove (`DESTROY`) an object or relationships, to add relationships between objects (`RELATE`), to save modifications (`SAVE`), to recreate the relations between existing objects (`REHASH`), and to restore the original data (`RESTORE`).

CHAPTER 8
EXPERIENCE WITH THE OBJECT FINDER

This chapter presents our experience in using the object finder to analyze programs. Our objective was to recover the design of programs by modularization using the algorithms outlined above.

An example used to illustrate the top-down analysis is the name cross-reference program. Another example of the top-down analysis is a simple algebraic expression evaluation program which was translated by hand from a C++ program with about 1,350 lines of code and 50 functions. Finally, an example of the bottom-up analysis is given using the name cross-reference program.

8.1 Example of the Top-down Analysis: Name Cross-reference Program

This section illustrates how to use the object finder in a top-down analysis of a system, its graphics user interface `xobject`, and some of the design recovery capabilities of the object finder.

The following example [26] is a name cross-reference program for "C" programs whose size statistics are given in Table 8.1. The sample program consists of five

Table 8.1. Statistics of the name cross-reference program.

	xref.c	xref_in.c	xref_out.c	xref_tab.c	xref_tab.h	Total
Lines of code	27	56	41	126	32	282
Global variables	0	1	0	0	0	1
Functions	1	2	1	6	0	10
Types (user-defined)	1	0	0	5	4	10

source files including one C header file and four program files. As explained above, there is a make file that contains a list of these files.

The top-down analysis approach was used with both the globals-based and types-based analyses; we choose to ignore "C" primitive types during the types-based analysis. The identified objects are shown in Figure 8.1. In Figure 8.1, a unique ID has been assigned to each identifier by the system automatically. For example, unique ID `xref_in~getachar~ch~58` (not shown in Figure 8.1) refers to variable `ch` declared in function `getachar` which is located in file `xref_in`. In the case of a "C" type, the corresponding unique ID has empty function and file components; for a type denoting a "C" array, the corresponding unique ID includes the type of the array elements followed by its dimension information, e.g., `~~char--~49` denotes a one-dimensional array of characters with unspecified size. In addition, the special "function" `___II` denotes the scope where external "C" functions and variables are defined. Finally, the unique IDs are accompanied by a code which denotes the kind of identifier (`R` for routines, `T` for types, and `G` for global data) and a copy of the identifier name; for user-defined types, additional information about the type is given such as the number of elements in each dimension for an array.

The name shown for each candidate object consists of a prefix `z#obj`, a code `G` for globals-based objects or `T` for types-based objects, the name of a global variable or type associated with the object, and a suffix consisting of `#` and a unique tag number. A candidate object is described by listing the set of routines, types, and global variables associated with the object during the analysis.

The identified objects are shown in Figure 8.2 as they appear using the graphics user interface `xobject`. At the top of Figure 8.2, there are a series of buttons which show the capabilities of this interface. There are buttons to set the working directory, to load the candidate objects and display their relationships, and to modify the

```
Object z#objTLINEPTR#3 is {
   ~~LINEPTR~16                              :(T)LINEPTR
   xref_tab~___II~make_linenode~138          :(R)make_linenode
}
Object z#objTWTPTR*#4 is {
   ~~WTPTR*~104                              :(T)WTPTR*
   xref_tab~___II~make_wordnode~140          :(R)make_wordnode
   xref_tab~___II~init_tab~102               :(R)init_tab
   xref_tab~___II~addword~64                 :(R)addword
   xref_tab~___II~add_lineno~61              :(R)add_lineno
   xref_out~___II~writewords~181             :(R)writewords
}
Object z#objTchar*#5 is {
   ~~char*~56                                :(T)char*
   xref_tab~___II~strsave~164                :(R)strsave
}
Object z#objTint*#6 is {
   ~~int*~69                                 :(T)int*
   xref~___II~main~137                       :(R)main
   xref_in~___II~getachar~93                 :(R)getachar
}
Object z#objT+undetermined-type#7 is {
   ~~int*~69                                 :(T)int*
   ~~char--~49                               :(T)char[]
   xref_in~___II~getword~98                  :(R)getword
}
Object z#objGlineno#2 is {
   xref_in~___II~lineno~130                  :(G)lineno
   xref_in~___II~getword~98                  :(R)getword
   xref_in~___II~getachar~93                 :(R)getachar
}
```

Figure 8.1. Candidate objects identified in the name cross-reference program

arrangement of the layout of the objects. The components of each identified object (data items, data types, and functions) are grouped in a light rectangle. The dark rectangles represent common components between two objects.

A user may modify the components of each object by changing the name, adding or deleting components, or moving components among objects (see Section 7.4 for a complete explanation of `xobject` capabilities). An example of the kind of modifications that can be made to the identified objects is to move a component from a candidate object and add it to another object such as removing routine `getword` from object `T+undetermined-type#7` and routine `getachar` from object `Tint*#6`. After such a modification, the user will recreate the relationships between the existing objects. The result is that the relationships between object `Glineno#2` and objects `Tint*#6` and `T+undetermined-type#7`, in Figure 8.2, will disappear because the common components between them have been eliminated.

The design recovery capabilities of the object finder can be appreciated in this example. The output of the object finder should be useful to the maintenance programmer since it provides high level information about the structure of the program. In this example, the object finder was able to identify key aspects of the original design of the cross-reference program by suggesting a decomposition of it into objects and graphically presenting the relationships among the objects. For example, the structure of the name cross-reference program in Figure 8.2 consists of four clusters[1] of candidate objects, the first consisting of objects `T+undetermined-type#7`, `Tint*#6`, and `Glineno#2`, the second consisting of `TWTPTR*#4`, the third consisting of `TLINEPTR*#3`, and the fourth consisting of `Tchar*#5`.

After viewing this suggested initial structure, the user is able to interactively modify the objects and their relationships. As explained above, two components

[1]These are different from Hutchens and Basili's hierarchical clusters.

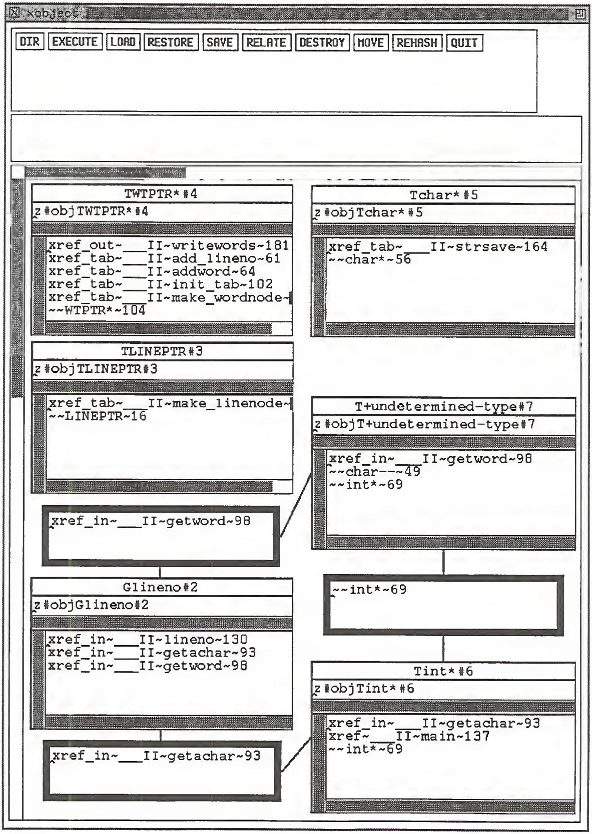

Figure 8.2. Candidate objects identified in the name cross-reference program displayed by `xobject`

Table 8.2. Statistics of the algebraic expression evaluation program.

	exprtst.c	function.c	symbol.c	state.c	function.h	symbol.h	state.h	Total
Lines of code	88	759	75	147	140	33	82	1324
Global variables	0	0	2	0	5	0	3	10
Functions	1	28	7	14	0	0	0	50
Types (user-defined)	3	6	2	0	1	1	1	14

were removed from two different candidate objects to provide a clearer design. The resulting objects after the modifications are shown in Figure 8.3.

In addition, the programmer is able to manually cluster related objects into a larger object. After the modifications concerning the removal of routines `getword` and `getachar` from objects `T+undetermined-type#7` and `Tint*#6`, respectively, we define five clusters of candidate objects in the cross-referencer program. The largest cluster, used for input and output in this program, consists of `T+undetermined-type#7` and `Tint*#6`. Each of the objects `TWTPTR*#4`, `TLINEPTR*#3`, and `Glineno#2`, constitutes a cluster of objects of size one and consists of an object that implements a given abstract data type. Finally, the object `Tchar*#5` constitutes a cluster of objects of size one which denotes an object that performs utility functions. The next step of refinement would disregard this routine from consideration by disabling it for the rest of the analysis.

8.2 Comparison with *C++* Classes: Algebraic Expression Evaluation Program

This section presents the results of study II mentioned in Chapter 5. In this section, a "C" program which is translated by hand from a "C++" program [44] is used to illustrate further design recovery capabilities of the object finder. The statistics of the translated version in "C" are listed in Table 8.2. The top-down analysis approach was executed, based on both globals-based and types-based analysis methods. Once

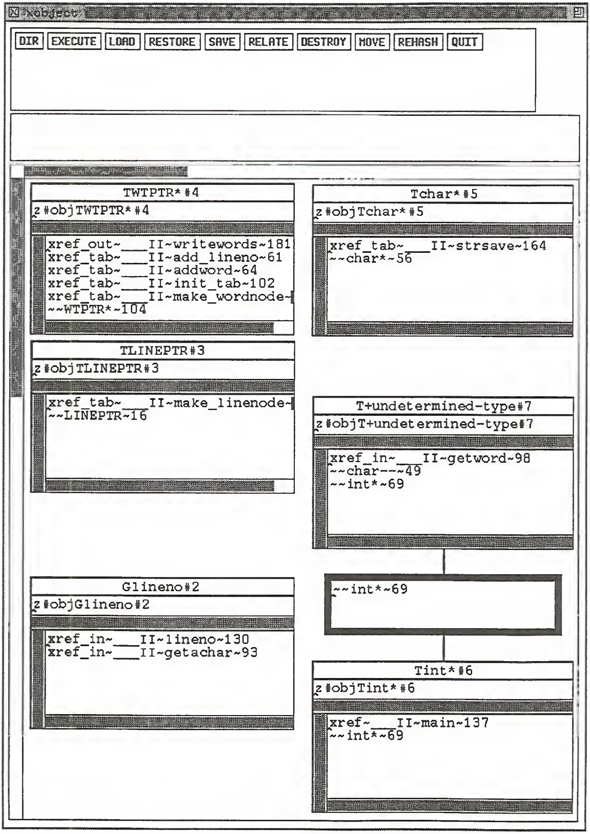

Figure 8.3. Modified candidate objects in the name cross-reference program displayed by `xobject`

Table 8.3. Comparison of candidate objects and "C++" classes.

Object in "C"	"C++" class	Explanation
G: `table_index#2`	`symbol_table`	All data and routines are matched.
G: `first#5`	`function`	Almost all data and routines are matched except one routine.

T: `NODE*#6`	`node`
	`variable`
	`plus`
	`divide`
	`constant`
	`multiply`
	`minus`
		The C++ classes `variable`, `plus`, `divide`, `constant`, `multiply`, and `minus` are subclasses of the C++ class `node`.

G: `expression#4` G: `paren_count#3`	`state0`	The class `state0` is divided into two candidate objects.

T: `char*#7`	`symbol_table`
	`function`
	`state0`
		This part was identified under user's request.

again, we choose to ignore the primitive types during the types-based analysis. The identified candidate objects are displayed in Figures 8.4 and 8.5.

In Table 8.3, we compared the identified candidate objects with the original "C++" classes.From Table 8.3, we observed that the "C++" class `symbol_table` perfectly matches the candidate object `Gtable_index#2`, and that the "C++" class `function` almost perfectly matches the candidate object `Gfirst#5` in terms of the set of routines in those candidate objects. Some differences between the derived objects and the original "C++" classes are explained below.

Object `TNODE*#6` is different from the "C++" classes listed in the third row of Table 8.3 because `node` is a *friend* class to the other classes in the "C++" version of the program and the translated "C" code merged those classes together as one combined type. In addition, `Gexpression#4` and `Gparen_count#3` are combined as one

```
Object z#objTNODE*#6 is {
   ~~NODE*~15                                :(T)NODE*
   function~___II~Variable_eval~140          :(R)Variable_eval
   function~___II~Plus_eval~104              :(R)Plus_eval
   function~___II~Node_eval~101              :(R)Node_eval
   function~___II~Multiply_eval~96           :(R)Multiply_eval
   function~___II~Minus_eval~94              :(R)Minus_eval
   function~___II~Function_precedence~82     :(R)Function_precedence
   function~___II~Function_delete_function   :(R)Function_delete_function
   function~___II~Function_checkandadd~68    :(R)Function_checkandadd
   function~___II~Function_build_tree~66     :(R)Function_build_tree
   function~___II~Function_add_operator~62   :(R)Function_add_operator
   function~___II~Function_add_operands~60   :(R)Function_add_operands
   function~___II~Echo_tree0~29              :(R)Echo_tree0
   function~___II~Divide_eval~22             :(R)Divide_eval
   function~___II~Construct_function~16      :(R)Construct_function
   function~___II~Constant_eval~12           :(R)Constant_eval
}
Object z#objTchar*#7 is {
   ~~char*~18                                :(T)char*
   symbol~___II~Symbol_table_get_index~136   :(R)Symbol_table_get_index
   symbol~___II~Symbol_table_add_variable~   :(R)Symbol_table_add_variable
   state~___II~State9_transition~128         :(R)State9_transition
   state~___II~State7_transition~126         :(R)State7_transition
   state~___II~State6_transition~124         :(R)State6_transition
   state~___II~State5_transition~122         :(R)State5_transition
   state~___II~State4_transition~120         :(R)State4_transition
   state~___II~State3_transition~118         :(R)State3_transition
   state~___II~State2_transition~116         :(R)State2_transition
   state~___II~State1_transition~114         :(R)State1_transition
   state~___II~State0_transition~112         :(R)State0_transition
   state~___II~State0_get_char~109           :(R)State0_get_char
   function~___II~Function_valid~85          :(R)Function_valid
   function~___II~Function_parenthesis~80    :(R)Function_parenthesis
   function~___II~Echo_tree~27               :(R)Echo_tree
}
```

Figure 8.4. Types-based candidate objects identified in the algebraic expression e-valuation program

```
Object z#objGtable_index#2 is {
   symbol~___II~table~284                      :(G)table
   symbol~___II~table_index~286                :(G)table_index
   symbol~___II~Symbol_table_get_value~138 :(R)Symbol_table_get_value
   symbol~___II~Symbol_table_get_index~136 :(R)Symbol_table_get_index
   symbol~___II~Symbol_table_clear~135      :(R)Symbol_table_clear
   symbol~___II~Symbol_table_add_variable~ :(R)Symbol_table_add_variable
   symbol~___II~Symbol_table_add_value~130 :(R)Symbol_table_add_value
   symbol~___II~Echo_symbol_table~26        :(R)Echo_symbol_table
   symbol~___II~Construct_symbol_table~20  :(R)Construct_symbol_table
}
Object z#objGparen_count#3 is {
   state~___II~paren_count~254               :(G)paren_count
   state~___II~State0_inc_paren~111          :(R)State0_inc_paren
   state~___II~State0_get_paren~110          :(R)State0_get_paren
   state~___II~State0_dec_paren~108          :(R)State0_dec_paren
}
Object z#objGexpression#4 is {
   state~___II~index~213                     :(G)index
   state~___II~expression~178                :(G)expression
   state~___II~State0_get_char~109           :(R)State0_get_char
   state~___II~Construct_state0~19           :(R)Construct_state0
   function~___II~Function_valid~85          :(R)Function_valid
}
Object z#objGfirst#5 is {
   function~___II~total_paren~292            :(G)total_paren
   function~___II~root~267                   :(G)root
   function~___II~queue~263                  :(G)queue
   function~___II~last~216                   :(G)last
   function~___II~first~193                  :(G)first
   function~___II~Function_precedence~82    :(R)Function_precedence
   function~___II~Function_parenthesis~80   :(R)Function_parenthesis
   function~___II~Function_ovldop~73         :(R)Function_ovldop
   function~___II~Function_delete_function :(R)Function_delete_function
   function~___II~Function_checkandadd~68   :(R)Function_checkandadd
   function~___II~Function_build_tree~66    :(R)Function_build_tree
   function~___II~Function_add_operator~62 :(R)Function_add_operator
   function~___II~Function_add_operands~60 :(R)Function_add_operands
   function~___II~Echo_tree0~29              :(R)Echo_tree0
   function~___II~Echo_queue~25              :(R)Echo_queue
   function~___II~Construct_function~16     :(R)Construct_function
}
```

Figure 8.5. Globals-based candidate objects identified in the algebraic expression evaluation program

group state0 in the original "C++" code. The reason is that paren_count is used as a counter of the variable expression. A semantic analysis is necessary to provide such a connection, which is not currently supported by the object finder. This will be explained for the future work of this dissertation. Finally, object Tchar*#7 consists of routines corresponding to several "C++" classes routines. Those routines which originally belonged in the "C++" classes symbol_table and function were removed, using the graphics user interface xobject, from object Tchar*#7, and added to objects Gtable_index#2, Gfirst#5, and Gexpression#4. After the modifications, the resulting object Tchar*#7 is part of the cluster of objects corresponding to the "C++" class state0 which already includes objects Gexpression#4 and Gparen_count#3.

We have compared the objects found by the object finder with those (classes) defined in the original "C++" program. There was great similarity between the two, demonstrating the design recovery capabilities of the object finder. We noted that object TNODE*#6 included the parent class node and its subclasses variable, plus, divide, constant, multiply, and minus from the original "C++" program.

Figure 8.6 illustrates the graphics representation of the objects modified as explained above. Object TNODE*#6 is related to candidate object Gfirst#5 because of the commonality between routines in Gfirst#5 (that update or reference their global variables) and routines in object TNODE*#6 (that manipulate their type, NODE). We argue that this cluster is a "natural cluster" for two reasons: firstly, except for two routines added by object Gfirst#5 to the cluster, the same set of routines appears in both objects, and secondly, the global variables (total_paren, root, queue, last, and first) in object Gfirst#5 support the manipulation of entities of type NODE in the program.

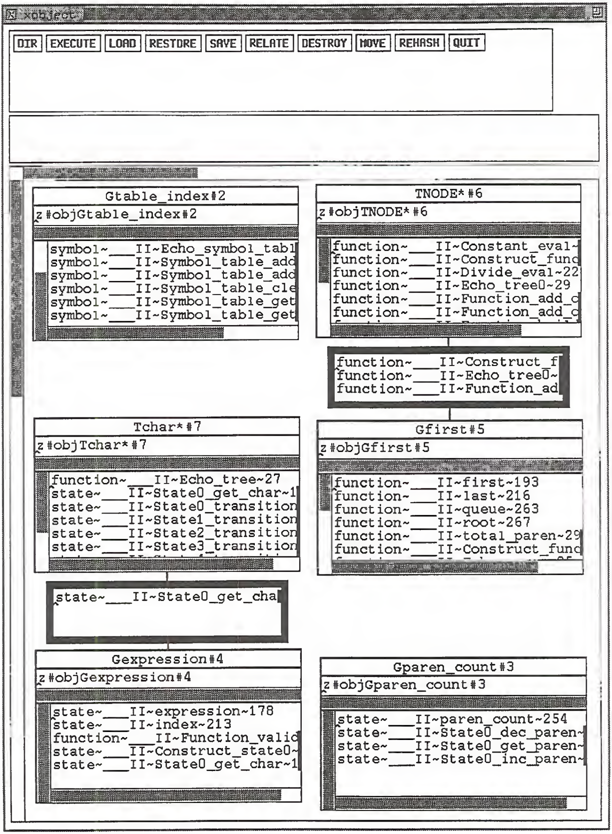

Figure 8.6. Candidate objects identified in algebraic expression evaluation program displayed by `xobject`

In addition, objects Tchar*#7 and Gexpression#4 are related because of the common routine State0_get_char in both objects. This cluster indicates the existence of a class state0 to the user. Object Gparen_count#3 is not part of this cluster even though it is part of the cluster related to class state0 in Table 8.3. Thus, the object finder signals the specialized nature of object Gparen_count#3 which constitutes a cluster of objects of size one. Finally, the object Gtable_index#2 constitutes a cluster of size one which denotes an object that performs symbol table management functions.

8.3 Example of the Bottom-up Analysis: Name Cross-reference Program

This section illustrates the use of the object finder in a bottom-up analysis of a system and some of the design recovery capabilities of the object finder. For this illustration we use the example from Section 8.1.

The bottom-up analysis approach was used with both globals-based and types-based analyses; we choose to ignore "C" primitive types during the types-based analysis. In this particular example, the initial goal of the user was to identify the candidate object that consists of the set of routines and data types which reference a global variable in the program; specifically, global variable lineno whose unique ID is xref_in~___II~lineno~130.

The process of bottom-up analysis begins with the user proposing a starting point: an initial candidate object which consists of several routines, types, or data items. The initial candidate object in this example consists of the global variable lineno, the set of routines which the user considers are suspect of manipulating the global variable lineno (initially only routine main is suspect), and the set of types of the return values and parameters of the suspect routines. Figure 8.7 shows this initial candidate object.

```
Object z#objmkfile#1 is {
   xref_in~___II~lineno~130                 :(G)lineno
   ~~char*~56                               :(T)char*
   xref~___II~main~137                      :(R)main
}
```

Figure 8.7. Initial candidate object defined by the user in the name cross reference program

```
Object z#objmkfile#1 is {
   xref_in~___II~lineno~130                 :(G)lineno
   ~~char*~56                               :(T)char*
   xref~___II~main~137                      :(R)main
   xref_in~___II~getword~98                 :(R)getword
   xref_in~___II~getachar~93                :(R)getachar
}
```

Figure 8.8. Extended candidate object resulting from the data–routine analysis

The next step of the process is that the user specifies the kind of refinement analysis of the initial candidate object. At first, the user chooses the data–routine analysis which consists of adding the set of all routines that use the global variable to the current set of routines in the object. After the analysis is automatically performed by the object finder, the extended candidate object is shown in Figure 8.8.

Figure 8.8 shows the extended candidate object. From this candidate object the user can determine that routines `getword` and `getachar` also use the specified global variable. The user will benefit from this knowledge by better understanding the set of routines which should be considered when making a change that affects the global variable.

Next, the user decide to perform two more iterations of the bottom-up analysis to build up his understanding about the affected routines when a change is made on global variable `lineno` and the data type "pointer to `char`." The first iteration

```
Object z#objmkfile#1 is {
   xref_in~___II~lineno~130                     :(G)lineno
   ~~int~4                                      :(T)int
   ~~int*~69                                    :(T)int*
   ~~char--~49                                  :(T)char[]
   ~~char*~56                                   :(T)char*
   ~~WTPTR~26                                   :(T)WTPTR
   ~~LINEPTR~16                                 :(T)LINEPTR
   xref~___II~main~137                          :(R)main
   xref_tab~___II~strsave~164                   :(R)strsave
   xref_tab~___II~make_wordnode~140             :(R)make_wordnode
   xref_tab~___II~make_linenode~138             :(R)make_linenode
   xref_tab~___II~addword~64                    :(R)addword
   xref_tab~___II~add_lineno~61                 :(R)add_lineno
   xref_out~___II~writewords~181                :(R)writewords
   xref_in~___II~getword~98                     :(R)getword
   xref_in~___II~getachar~93                    :(R)getachar
}
```

Figure 8.9. Final candidate object

consists of a type–routine analysis which involves adding to the current set of routines in the object, the set of all routines that use the types in the object as formal parameters or return values. The second iteration consists of a routine–type analysis which adds to the current set of types in the object, the set of all types used as formal parameters or return values by the routines in the object after the last analysis. The resulting candidate object is shown in Figure 8.9.

Figure 8.9 shows the extended candidate object consisting of routines affected by changes to global variable `lineno` and those routines manipulating an abstract data type with type pointer to `char`.

Some of the design recovery capabilities of the object finder are observed from this example. The user begins with a starting point, an initial candidate object, and builds an agglomerative understanding of the dependencies between the system components by adding routines, types, or data to this initial candidate object according to the analysis performed. During the analysis, the user is aided by the connections

between components which are drawn at each iteration of the analysis. If the user desires, the result shown in Figure 8.9 can be further analyzed using the top-down analysis.

CHAPTER 9
CONCLUSIONS AND FURTHER STUDY

Identifying object-like features in code written in conventional programming languages would seem to be a useful step in applying object-oriented methods to the reengineering and maintenance of old code. This dissertation has described a strategy based on persistent data and data type for recovering the underlying system structure of an existing program by identifying the data, structures, and their relationships. The approach's output is a modularization of a program that consists of the collection of "objects" found in the program.

We believe that tools like the object finder can be a very useful aid in maintaining and reusing old code. Useful existing programs may be discarded because there is no easy way of understanding their structure. But if the original programmer made good use of modularity and abstract data typing techniques, tools like the object finder can help to recover these aspects of the design and thus extend the life and value of the code.

The object finder is somewhat analogous to other software clustering methods that have been proposed [14, 36] but we believe that it is unique in searching for a particular kind of cluster, one that is similar to an abstract data type or to an object [5]. It represents a narrow but useful kind of software design recovery, unlike many of the more sophisticated and much more complicated general methodologies that have been proposed. While the object finder is obviously only a beginning, we believe that it illustrates the potential for the development of many kinds of interactive design recovery tools.

This dissertation has also presented the experience of using the object finder algorithms and an evaluation of the analysis by the object finder. This evaluation included an examination of the results of this approach of system modularization. Two studies were used to evaluate the identified objects. In study I, we compared the groups identified using the object finder (candidate objects) with those groups identified using Hutchens and Basili's hierarchical clustering [14] (hierarchical clusters). The comparison was based on the complexity of the two partitionings resulting from each clustering technique. The results of the evaluation included a new sets of complexity metrics factors: the inter-group complexity metrics factors, that measured the complexity of the interface of a group, the intra-group complexity metrics factors, that measured the internal complexity of a group, and the intra-group strength factors, that measured the "functional relatedness" of the types of formal parameters or return values in the routines in a group. These factors measured the complexity (similar to coupling and cohesion) of a given group in a partitioning. The newly defined metrics factors were validated by showing that the complexity metrics factors are sensitive to changes in the structure of programs. Three "C" programs were used as test cases for this study. The conclusions of this study indicated that the object finder approach results in a modularization of a program which exhibited significantly less inter-group complexity and moderately less intra-group complexity, when compared to the modularization using hierarchical clustering techniques. The comparison did not include intra-group strength factors since hierarchical clustering does not consider the characteristics measured by these factors during the modularization.

In study II, we compared the identified objects in a program with the classes found in the object-oriented version of the same program. To ensure that the program versions were functionally equivalent, an object-oriented program (written in "C++") was translated into an equivalent non object-oriented version of the program. The

results of study II indicated that the identified objects by the object finder closely match the corresponding object-oriented language classes. For instance, there was a very close match between the identified objects using the Globals-based Object Finder algorithm and the "C++" classes. Furthermore, the identified objects illustrated the design recovery capabilities of the object finder by providing a greater level of refinement of the groups in a program than the classes found in the "C++" program. For example, when several identified objects correspond to a single "C++" class, it represents the specialized nature of the functionality of the set of routines grouped into the identified objects. On the other hand, the "C++" version of the program merged all the routines together in a class.

The main contributions of this research are a new program modularization approach based on type binding information, an efficient approach of object identification in non object-oriented languages, and a new set of complexity metrics factors to evaluate the results of the approach. The evaluation studies in this dissertation provided the guidelines for an experiment to further evaluate these object identification methods.

Future improvements to the current prototype interactive software tool, which implements these object identification methods for the "C" language, will allow us to experiment with finding objects in samples of industrial-size code. Our intention is to collect data on the user's decisions and his reasons for making them so that improved heuristics or knowledge-based methods can be developed to improve the process. Further evaluation of the object finder algorithms consists of an statistical experimentation using production-size programs such as the GNU "C" compiler gcc.

In addition, further study is needed to determine additional characteristics that could be used to identify the object-like features of conventional programming languages based on object-oriented design principles [5]. These principles include classification of the objects in the application, organization of the objects, and identification of operations on the objects. The knowledge required to abstract the object-like features of programs includes semantic knowledge about the program; for example, the fact that a variable is used as a "counter" of iterations on a given data structure is determined from a semantic analysis. One solution is collecting semantic knowledge about object-oriented programs [45] as an initial knowledge base about the program. Moreover, other knowledge useful in identifying object-like characteristics consists of a detailed analysis of the internals of routines in the system such as data flow analysis of the routine. The future study will also extend the object finder to consider the inheritance characteristics of objects when identifying objects. A possible approach to reveal the inheritance characteristics between identified objects is to record the existence of relationships (defined by common routines) among objects. The common routines indicate that a common functionality is shared by the sharing objects. This commonality could be represented as a "generalized object" with each of the sharing objects containing routines that specialize the generalized object.

In the future research, we propose to extend the evaluation criteria of the object finder to other metrics that measure factors such as the degree of modularization [7] and the "object-orientedness" of the abstracted design. The latter factors would be based on object-oriented concepts and principles. The complexity metrics factors defined in this dissertation measured several design qualities of objects based on their cohesion and coupling, including information hiding, encapsulation (state and operations), and abstraction (abstract class). Furthermore, two other design qualities of the identified objects are inheritance and polymorphism which could be used to

measure their re-usability. Future research will define the primitive metrics factors which measure these design qualities.

An efficient approach of object identification in non object-oriented languages has been presented. The proposed Globals-based Object Finder algorithm has polynomial time complexity in terms of the program length as measured by the number of global variables and the number of routines in the program. Since the number of global variables is usually small compared to the number of routines in the program, the method is practical in most software applications. An improvement to the performance of Algorithm 1 was outlined in Chapter 4 and the graph construction step which results in almost linear time complexity in terms of the number of routines. The proposed Types-based Object Finder algorithm has linear complexity in terms of the number of routines and the number of (abstract data) types.

Module-level understanding of the system structure is mainly needed when a maintainer is coming to grips with a system for the first time. This understanding will allow him to sort out the components of a system and perceive the overall architecture of the system. In addition, he may build a framework to assists him in making sense of the more detailed information he will acquire during the maintenance activity. Thus, the need for this understanding is clear. The object finder methods assists the programmer in attaining some of this high level understanding. We have proved that our methods are at least as good as current practices for system modularization and high level design knowledge abstraction. They provide extra help in considering type information to provide this knowledge. Even though, this is of great help, the job is not complete yet. We foresee more complete techniques of module-level program understanding using our methods as well as object-oriented principles for this endeavor.

APPENDIX A
OBJECT FINDER PROTOTYPE USER'S MANUAL

A.1 Introduction

This is a user's manual for a GNU Emacs-based [1] object finder prototype developed at the Software Engineering Research Center, Department of Computer and Information Sciences, University of Florida. It is currently running on a Unix BSD 4.3 environment with GNU Emacs 18.55 installed. We expect no problem to install it to other Unix systems as long as a GNU Emacs version 18 is installed.

This prototype is a design recovery tool. The goal of the object finder is to aid an analyst or maintainer in recovering certain kinds of design principles and system structures from source code. The object finder looks for clusters of data, structures and routines, that are analogous to the objects and object classes of object-oriented programming.

The programming language supported is ANSI "C". "Emacs-based" means that all Emacs features, such as editing several files simultaneously, opening multiple windows on the same document, and undoing mistakes, ..., etc, are still available. It should also be noted that the prototype assumes and expects syntactically correct programs before analysis may begin.

Section A.2 of this manual will cover the necessary setup and operational procedures. Also included are definitions of the system buffers and system files in Section A.3.

A.2 Operation

In this section, we explain how to operate the system in detail.

A.2.1 Basic Setup Commands

1. Invoke GNU Emacs:

```
% emacs
```

2. Initialize this prototype:

```
M-x erds
```

A.2.2 Object Finder Analysis

A user now may generate a file describing the system to be analyzed. Currently, the prototype analyzes a "C" program that consists of a collection of source files and a `mkfile` file which describes how the program is constructed. The created files must be saved before invoking this prototype. This prototype is invoked by using:

```
ESC ESC ESC o
```

The system will prompt with `From:`. Type the file name, e.g. `mkfile`. An Emacs buffer is created for each data file.

Menu-driven commands are available to perform the analysis.

A menu will be displayed in the minibuffer (at the bottom of this screen) as follows:

```
T:Top-down analysis B:Bottom-up analysis?
```

The meaning related to each option follows:

- `T` or `t`: performs top-down analysis.
- `B` or `b`: performs bottom-up analysis.

The first step of the bottom-up analysis is to propose a starting point – one or more routines, types, or global variables to be part of an initial candidate object. For this, the system asks for the unique ID of the global variables, types, or routines until `done` is entered.

A.2.3 Display Analysis Results and Identified Objects

Menu-driven commands are available to display the results of the globals-based and types-based analysis. The user defines several settings to control the appearance of the output as follows:

- ```
  Show graph G:Current global name length [CURRENT-VALUE]: ,
                              set _global-name-length to value:
  ```

  Determines the width in characters of the global name column in graph G.

- ```
  Show graph G:Current global name length [CURRENT-VALUE]: ,
                              set _global-name-length to value:
  ```

  Determines the width in characters of the type name column in matrix R.

- ```
  Show graph G:Current column width is [CURRENT-VALUE]: ,
                                 set _column-width to value:
  ```

  Determines the width in characters of a column in graph G or matrix R.

In any of the choices above, just press return to choose the default `CURRENT-VALUE`.

After the analysis results are displayed, the user is confirmed to display the identified candidate objects from the current candidate object:

```
Show candidate objects?(y/n)
```

Answer `y` to display candidate objects; otherwise, answer `n`.

A.2.3.1 Top-down analysis and display

The following settings are specified by the user during the top-down analysis:

```
Globals analysis for object OBJECT-NAME ?
```
```

Answer `y` if globals-based analysis is performed on the program; otherwise, answer `n`.

`Types analysis for object OBJECT-NAME ?`

Answer `y` if types-based analysis is performed on the program; otherwise, answer `n`.

Both settings are displayed back to the user and the system waits for confirmation:

`Confirm top down settings above for object OBJECT-NAME ?`

Answer `y` if the user agrees with the settings; otherwise, answer `n`.

During types-based analysis the user is asked whether to consider the primitive types in the analysis.

`Type analysis grouping with basic types?`

Answer `y` if consider primitive types; otherwise, answer `n` to ignore primitive types.

The user may further refine the analysis results, in a top-down fashion, when the top-down analysis menu is displayed:

```
Continue(c);disable globals(d),routines(f),types(y);exit(e);quit(q);
    xobject(x)?
```

The meaning related to each option follows:

- `c`: Continues top-down analysis and invokes the object selection menu.
- `d`: Disables some global variables from the analysis, and prompt the user for the names (unique ID) of the global variables until `done` is entered.
- `f`: Disables some routines from the analysis, and prompt the user for the names (unique ID) of the routines until `done` is entered.
- `y`: Disables some types from the analysis, and prompt the user for the names (unique ID) of the types until `done` is entered.
- `e`: Exits object finder saving the results in files.

- `q`: Quits object finder without saving the results in files.

- `x`: Invokes the X Window-based graphics user interface `xobject` [31].

To continue the top-down analysis, the object selection choice (after `c` has been typed) will prompt a menu line as follows:

```
Choose object from subobjects(l),any object(t);repeat object(r);
read object(o)?
```

The meaning related to each option follows:

- `l`: Chooses the candidate object from the identified candidate objects of the current candidate object.

- `t`: Chooses the candidate object from all the identified candidate objects in the complete system (a tree).

- `r`: Repeats analysis of the current candidate object.

- `o`: Read the candidate objects of the current candidate object from files. Usually, after `xobject` has modified and saved these candidate objects into files.

For choices `l` and `t` above, specify the chosen candidate object `OBJECT-NAME` followed by any file extension in Section A.3.1.

A.2.3.2 Bottom-up analysis and display

The following bottom-up settings are set by the user:

```
Data-routine (dr), routine-data (rd), type-routine (tr),
    routine-type (rt)?
```

The meaning related to each setting follows:

- `dr` or `DR`: adds to the current object the set of all routines that use global variables in the object.
- `rd` or `RD`: adds to the current object the set of all global variables used by routines in the object.
- `tr` or `TR`: adds to the current object the set of all routines that use types in the object for formal parameters or return values.
- `rt` or `RT`: adds to the current object the set of all types used as formal parameters or return values by routines in the object.

Then, the following settings are set by the user during bottom-up analysis:

`Globals analysis for object OBJECT-NAME ?`

Answer `y` if globals-based analysis is performed on the program; otherwise, answer `n`.

`Types analysis for object OBJECT-NAME ?`

Answer `y` if types-based analysis is performed on the program; otherwise, answer `n`.

All settings are displayed back to the user and the system waits for confirmation:

`Confirm bottom-up settings above for object object-name ?`

Answer `y` if the user agrees; otherwise, answer `n`.

During types-based analysis the user is asked whether to consider the primitive types in the analysis.

`Type analysis grouping with basic types?`

Answer `y` if consider primitive types; otherwise, answer `n` to ignore primitive types.

The user may further refine the analysis, in a bottom-up fashion, when the bottom-up analysis menu is displayed:

```
Repeat(r);disable globals(d),routines(f),types(y);exit(e);quit(q);
    XObject(x)?
```

The meaning related to each option follows:

- `r`: Repeats analysis of the current candidate object.
- `d`: Disables some global variables from the analysis, and prompt the user for the names (unique ID) of the global variables until `done` is entered.
- `f`: Disables some routines from the analysis, and prompt the user for the names (unique ID) of the routines until `done` is entered.
- `y`: Disables some types from the analysis, and prompt the user for the names (unique ID) of the types until `done` is entered.
- `e`: Exits object finder saving the results in files.
- `q`: Quits object finder without saving the results in files.
- `x`: Invokes the X Window-based graphics user interface `xobject`. This feature is currently disabled during bottom-up analysis.

A more extensive explanation on the GNU Emacs commands is given in the GNU Emacs Manual [40].

A.3 Buffer Structure and Files

A.3.1 System Buffer Structure

1. `OBJECT-NAME.src` — the `mkfile` file.
2. `OBJECT-NAME.stb` — the symbol table (internal representation) of the candidate object.
3. `OBJECT-NAME.xrf` — the `acx` cross-reference information for the candidate object.

4. `OBJECT-NAME.rou` — the routines in the candidate object.

   The routines are identified by their unique ID and are saved one-per-line.

5. `OBJECT-NAME.typ` — the types in the candidate object.

   The types are identified by their unique ID and are saved one-per-line.

6. `OBJECT-NAME.gbl` — the global variables in the candidate object.

   The global variables are identified by their unique ID and are saved one-per-line.

7. `OBJECT-NAME.pv` — set of routines which directly use variable x: P(x).

   Contains lines of the format:

   ```
   (x P(x))
   ```

   for each global shared variable x, `x` is the unique ID of variable x.

8. `OBJECT-NAME.pe` — set of edges between global variables in graph G from the globals-based analysis. Contains lines of the format:

   ```
   (x1 x2 (P(x1) intersection P(x2)))
   ```

   where `x1` and `x2` are unique ID's of global variables and `P(x1) intersection P(x2)` is the set of routines which directly use both x1 and x2.

9. `OBJECT-NAME.co` — set of candidate objects generated by the globals-based and types-based analysis. Contains lines of the format:

   ```
   "OBJECT-NAME"
   ```

10. `OBJECT-NAME.to` — types ordering in the types hierarchy. Contains lines of the format:

`(type subtype-attribute)`

where `type` is the unique ID of a type that is a super-type of the types listed in `subtype-attribute`.

11. `OBJECT-NAME.r` — set of entries in the matrix R from the types-based analysis.

    Contains lines of the format:

    `(routine type R(routine,type))`

12. `OBJECT-NAME.PE` — the graph G from the globals-based analysis.

    This buffer is displayed.

13. `OBJECT-NAME.R` — the matrix R from the types-based analysis.

    This buffer is displayed.

14. `OBJECT-NAME.CO` — set of candidate objects generated by the analysis.

    This buffer is displayed.

15. `OBJECT-NAME.Xbj` — The `xobject` process output buffer.

16. `OBJECT-NAME.log` — the system run time information. The information consists of the current time in string format of an event in a function.

A.3.2 System Files

1. analysis.el – Perform the top-down and bottom-up analysis and the globals-based analysis.

2. buffers.el – Buffer control library.

3. demo.el – Demonstration functions.

4. erds-interface.el – Dump output functions.
5. erds-manual.el – Menu-driven support library.
6. erds.el – ERDS mode.
7. init.el – System initialization.
8. layout.el – Emacs window layout control functions.
9. ofind.el – Object finder main driver.
10. parser2.el – Parser for the cross-reference information from `acx`.
11. plists.el – Property list functions.
12. sets.el – Set control library.
13. show.el – Analysis results and candidate objects display functions.
14. sys-log.el – System run-time log.
15. types.el – Perform the types-based analysis.
16. xref2.el – Main driver for the parser of the cross-reference information from `acx`.
17. .emacs – An Emacs initial setup file.
18. .emacs_ – A directory to hold all Emacs initialization files.
19. erds.texinfo – This user's manual.

REFERENCES

[1] J. Ambras and V. O'Day, "MicroScope: A Knowledge-Based Programming Environment," *IEEE Software*, Vol. 5, No. 3, May 1988, pp. 23–27.

[2] V. R. Basili and A. J. Turner, "Iterative Enhancement: A Practical Technique for Software Development," *IEEE Trans. on Software Engineering*, Vol. SE-1, No. 4, March 1975, pp. 390–396.

[3] L. A. Belady and C. J. Evangelisti, "System Partitioning and Its Measure," *Journal of Systems and Software*, Vol. 2, No. , 1981, pp. 23–29.

[4] T. J. Biggerstaff, "Design Recovery for Maintenance and Reuse," *IEEE Computer*, Vol. 22, No. 7, July 1989, pp. 36–50.

[5] G. Booch, "Object-Oriented Development," *IEEE Trans. on Software Engineering*, Vol. SE-12, No. 2, Feb. 1986, pp. 211–221.

[6] L.L. Constantine and E. Yourdon, *Structured Designs*, Englewood Cliffs, New Jersey, Prentice–Hall, 1979.

[7] S.D. Conte, H.E. Dunsmore, and V.Y. Shen, *Software Engineering Metrics and Models*, Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1986.

[8] T. J. Emerson, "A Discriminant Metric for Module Cohesion," *Proc. of the Seventh International Conference on Software Engineering*, Orlando, Florida, March 26–29, 1984, pp. 294–303.

[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Trans. on Programming Languages and Systems*, Vol. 9, No. 3, July 1987, pp. 319–349.

[10] M. Gondran and M. Minoux, *Graphs and Algorithms*, John Wiley & Sons, Interscience Publication, New York, 1984.

[11] S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Trans. on Software Engineering*, Vol. SE-7, No. 5, September 1981, pp. 510–518.

[12] M. S. Hecht, *Flow Analysis of Computer Programs*, Elsevier North-Holland, Inc., New York, 1977.

[13] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *Proc. of the ACM SIGPLAN 88, Conference on Programming Language Design and Implementation,* Atlanta, Georgia, June 22–24, 1988, pp. 35–46.

[14] D. Hutchens and V. Basili, "System Structure Analysis: Clustering with Data Bindings," *IEEE Trans. on Software Engineering*, Vol. SE-11, No. 8, August 1985, pp. 749–757.

[15] D. Kafura and G.R. Reddy, "The Use of Software Complexity Metrics in Software Maintenance," *IEEE Trans. on Software Engineering*, Vol. SE-13, No. 3, March 1987, pp. 335–343.

[16] G. E. Kaiser, P. H. Feiler, and S. S. Popovich, "Intelligent Assistance for Software Development and Maintenance," *IEEE Software*, Vol. 5, No. 3, May 1988, pp. 40–49.

[17] D.E. Knuth, *The Art of Computer Programming*, 2nd. Edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1973.

[18] T. Korson and J. D. McGregor, "Understanding Object-oriented: A Unifying Paradigm," *Communications of ACM*, Vol. 33, No. 9, September 1990, pp. 40–60.

[19] W. Kozaczynski and J. Ning, *SRE: A Knowledge-Based Environment for Large-Scale Software Re-engineering Activities*, Arthur Andersen & Co., CSTaR Discussion Note 014-88, September 1988.

[20] B. Lewis, *GNU Emacs Lisp Programmer's Manual*, Free Software Foundation, Cambridge, Massachusetts, 1990.

[21] B. Li, *Identifying Software System Structure Using Data Binding and Type Binding*, Masters Thesis, Computer and Information Sciences Department, University of Florida, August 1991.

[22] S. S. Liu and N. Wilde, "Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery," *Proc. Conf. on Software Maintenance*, San Diego, California, November 1990, pp. 266–271.

[23] Y. S. Maarrek "On the Use of Cluster Analysis for Assisting Maintenance of Large Software Systems," *Proc. of the Third IEEE Israel Conference on Computer Systems and Software Engineering*, Tel Aviv, Israel, June 1988, pp. 178–186.

[24] M. Marcotty and H. F. Ledgard, *Programming Language Landscape*, Second Edition, Science Research Associates, Inc., Chicago, Illinois, 1986.

[25] T. J. McCabe, "A Complexity Measure," *IEEE Trans. on Software Engineering*, Vol. SE-2, No. 4, December 1976, pp. 308–320.

[26] L. H. Miller and A. E. Quilici, *Programming in C*, John Wiley & Sons, Inc., New York, 1986, pp. 309–313.

[27] G. J. Myers, *Composite/Structured Design*, Van Nostrand Reinhold, New York, 1978.

[28] A. Nye, *The Definitive Guides to the X Window System: Xlib Programming Manual, Version 11*, Sebastopol, California, O'Reilly and Associates, Inc., 1988.

[29] P.E. Presson, J. Tsai, T.P. Bowen, J.V. Post, and R. Schmidt, *Software Interoperability and Reusability, Guidebook for Software Quality Measurement*, Rome Air Development Center, Griffiss Air Force Base, New York, Rep. RADC–TR83–174, July 1983.

[30] C.V. Ramamoorthy, W.T. Tsai, T. Yamaura, and A. Bhide, "Metrics Guided Methodology," *Proc. COMPSAC 85*, October 1985, pp. 111–120.

[31] C. Reddecliff, *Graphical User Interface for an Object Finder*, Senior Project, Computer and Information Sciences Department, University of Florida, August 1990.

[32] C. Rich and R. Waters, "The Programmer's Apprentice: A Research Overview," *IEEE Computer*, Vol. 21, No. 11, November 1988, pp. 10–25.

[33] J. A. Ricketts, J. C. DelMonaco and M. W. Weeks, "Data Reengineering for Application Systems," *Proc. Conf. on Software Maintenance*, Miami, Florida, October 1989, pp. 174–179.

[34] H. Schildt, *C: The Complete Reference*, Osborne McGraw-Hill, Inc., Berkeley, California, 1987.

[35] R. W. Schwanke, R. Z. Altucher, and M. A. Platoff, "Discovering, Visualizing, and Controlling Software Structure," *Proc. Fifth International Workshop on Software Specification and Design*, May 1989, pp. 147–150.

[36] R. W. Schwanke and M. A. Platoff, "Cross References are Features," *Proc. Second International Workshop on Software Configuration Management*, Princeton, New Jersey, 1989, pp. 86–95.

[37] M.L. Shooman, *Software Engineering: Design, Reliability, and Management*, 3rd. Ed., McGraw-Hill, Inc., New York, 1983.

[38] J.D. Smith, *Design and Analysis of Algorithms*, PWS-KENT Publishing Company, Boston, Massachusetts, 1989.

[39] I. Sommerville, *Software Engineering*, 3rd. Ed., Addison-Wesley Publishing Co., Reading, Massachusetts, 1989.

[40] R. Stallman, *GNU Emacs Manual*, Free Software Foundation, Cambridge, Massachusetts, 1990.

[41] W.P. Stevens, G. J. Myers, and L. L. Constantine, "Structural Design," *IBM Syst. J.*, Vol. 13, No. 2, 1974, pp. 115–139.

[42] R. Tarjan, "Depth-first Search and Linear Graph Algorithms," *SIAM J. on Computing*, Vol. 1, No. 2, June 1972, pp. 146–160.

[43] M. Weiser, "Program Slicing," *IEEE Trans. on Software Engineering*, Vol. SE-10, No. 4, July 1984, pp. 352–357.

[44] R. S. Wiener and L. J. Pinson, *An Introduction to Object-oriented Programming and C++*, Addison-Wesley, Reading, Massachusetts, 1988.

[45] N. Wilde and R. Huitt, *Maintenance Support for Object Oriented Programs*, SERC-TR-47-F, University of Florida/Purdue University, March 1991.

[46] N. Wilde, R. Huitt, and S. Huitt, "Dependency Analysis Tools: Reusable Components for Software Maintenance," *Proc. Conf. on Software Maintenance*, Miami, Florida, October 1989, pp. 126–131.

[47] P.E. Winston and B.K. Horn, *Lisp*, 2nd. Edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[48] S. S. Yau and J. S. Collofello, "Some Stability Measures for Software Maintenance," *IEEE Trans. on Software Engineering*, Vol. SE-6, No. 6, November 1980, pp. 545–552.

[49] S. S. Yau and S. S. Liu, "A Knowledge-Based Software Maintenance Environment," *Proc. COMPSAC 86*, October 1986, pp. 72–78.

[50] S. S. Yau and S. S. Liu, *Some Approaches to Logical Ripple Effect Analysis*, SERC-TR-24-F, University of Florida/Purdue University, September 1988.

[51] W. Zage and D. Zage, *Relating Design Metrics to Software Quality: Some Empirical Results*, SERC-TR-74-P, Purdue University, West Lafayette, Indiana, May 1990.

BIOGRAPHICAL SKETCH

Roger M. Ogando was born on December 11, 1961 in Santo Domingo, Dominican Republic. He received a B.S. degree in industrial engineering from the Instituto Tecnológico de Santo Domingo (INTEC), Santo Domingo, Dominican Republic, in October 1981. Mr. Ogando also received an M.S. degree in computer sciences from the University of Dayton, Dayton, Ohio, in 1985. He is a past recipient of the Fulbright Fellowship and of a PRA Fellowship from the Organization of American States in 1983 and 1987, respectively. He began his Ph.D. program in computer and information sciences at the University of Florida in the Fall 1985. He is also student member of the Association for Computing Machinery (ACM) and the Institute of Electrical and Electronics Engineers (IEEE) since 1984.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Stephen S. Yau, Chairman
Professor of Computer and
Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Stephen M. Thebaut, Cochairman
Assistant Professor of Computer and
Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Randy Chow

Randy Chow
Professor of Computer and
Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Justin Graver
Assistant Professor of Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Jack Elzinga
Professor of Industrial and Systems Engineering

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

August, 1991

Herbert A. Bevis
for Winfred M. Phillips
Dean, College of Engineering

Madelyn M. Lockhart
Dean, Graduate School